

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



# 数据库事务 处理的艺术

## 事务管理与并发控制

李海翔 等著

The Art of Database Transaction Processing:  
Transaction Management and Concurrency Control

- 中国计算机学会（CCF）常务理事、数据库专委会主任、数据库领域著名专家、中国人民大学杜小勇教授亲自作序推荐
- 作者有近20年数据库内核研发经验，曾是Oracle公司MySQL全球开发组核心成员，现在是腾讯的T4级专家



机械工业出版社  
China Machine Press





## 内容简介

作者有近20年数据库内核研发经验，曾是Oracle公司MySQL全球开发组核心成员，现在是腾讯的T4级专家。数据库领域的泰斗杜小勇老师亲自为本书作序，数据库学术界的知名学者张孝博士（中国人民大学）、卢卫博士后（中国人民大学）、彭煜玮博士（武汉大学），以及数据库工业界的知名专家盖国强和姜承尧等也对本书给予了极高的评价。

全书共12章，首先介绍数据库事务管理与并发控制的基础理论和工作机制，然后再从工程实践的角度对比和分析了4个主流数据库的事务管理与并发控制的实现原理，最后通过源代码分析了PostgreSQL和MySQL在事务管理与并发控制上的技术架构与设计思想。

### 第一篇（第1章和第2章）事务管理与并发控制基础理论

对数据库事务管理和并发控制的基础理论、核心技术和工作原理进行了讲解，包括数据库事务处理技术的范围、数据的异常现象及成因、事务模型、并发访问控制技术以及隔离性等。

### 第二篇（第3~6章）事务管理与并发控制应用实例研究

以Informix、Oracle、PostgreSQL和MySQL/InnoDB等主流数据库系统为例，对它们的事务管理和并发控制的实现技术、工作原理以及原理背后的设计思想进行了深度分析和对比。

### 第三篇（第7~9章）PostgreSQL事务管理与并发控制源码分析

首先对PostgreSQL事务处理技术的架构、层次、设计思想、相关数据结构和实现原理进行了深入系统的分析，然后从功能角度对PostgreSQL的事务模型、并发控制、一致性、隔离性以及其所使用的SS2PL、MVCC、SSI等技术做了深入的讲解。

### 第四篇（第10~12章）InnoDB事务管理与并发控制源码分析

首先对MySQL/InnoDB事务处理技术的架构、层次、设计思想、相关数据结构和实现原理进行了深入系统的分析，然后从功能角度对MySQL/InnoDB的事务模型、并发控制、一致性、隔离性以及其所使用的SS2PL、MVCC等技术做了深入的讲解。







华章科技  
HZBOOKS | Science & Technology









数据库  
技术丛书

# 数据库事务 处理的艺术

## 事务管理与并发控制

---

The Art of Database Transaction Processing:  
Transaction Management and Concurrency Control

---

李海翔 冯毅 范鹏程 著



机械工业出版社  
China Machine Press





## 图书在版编目 (CIP) 数据

数据库事务处理的艺术：事务管理与并发控制 / 李海翔等著，—北京：机械工业出版社，2017.10

(数据库技术丛书)

ISBN 978-7-111-58235-9

I. 数… II. 李… III. 关系数据库系统 IV. TP311.138

中国版本图书馆 CIP 数据核字 (2017) 第 245072 号

## 数据库事务处理的艺术：事务管理与并发控制

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：何欣阳

责任校对：殷虹

印刷：北京诚信伟业印刷有限公司

版次：2017 年 10 月第 1 版第 1 次印刷

开本：186mm×240mm 1/16

印张：33.5

书号：ISBN 978-7-111-58235-9

定价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东







## 推荐序一

海翔在数据库管理系统领域的第二本著作《数据库事务处理的艺术：事务管理与并发控制》马上就要出版了，他邀请我作序，我没有犹豫就欣然答应了。事后，我自己都觉得奇怪为什么会这么痛快，但细细想来，还是有充分理由的。这个序得写！

首先，我对在数据库核心技术领域长期辛勤耕耘的人表示尊敬。数据库是信息系统的基础和核心，对数据库实现技术是否真正掌握关系到我国在信息技术核心领域的自主可控战略是否能顺利实现。大家都知道，长期以来我国的信息产业大而不强，信息化成本居高不下，信息安全受到威胁。数据库就是基础软件中的最重要的部分之一。因此，国家从十五开始就对这个领域的研究开发进行了持续的支持。先是在 863 计划下设立“数据库重大专项”，后来又在国家中长期科技发展规划中设立了“核心电子器件、高端通用芯片和基础软件产品（简称核高基）”项目，对这个方向的研究开发工作给予引导。但是，由于国外数据库巨头已经形成市场垄断，短期间内企业要在这个领域赢利几无可能，因此，一些有实力的大企业都不愿意投身到这个领域来，形成了长期都由一些小公司在苦苦挣扎的窘境。人才能否“引得来、留得住、长得好”，是这些技术型小公司普遍遇到的问题。因此，我对长期坚持在这个领域的从业人员始终抱有好感，只要可能就愿意帮助他们。

其次，从我们教学的经验看，要深入理解数据库实现，非常需要解读开源系统实现代码的图书作为学习的参考。数据库系统庞杂，要弄懂系统实现的代码需要锲而不舍的精神。海翔阅读了 PostgreSQL 和 MySQL（InnoDB）等几个开源系统的代码，并根据自己的理解进行了解读，通过对比不同的实现能帮助读者深入了解事务的概念和实现技术。事务管理是数据库系统的核心技术，有一句话形容事务管理“好懂难做”，意思是要理解这些概念并不难，但是要实现起来，还是很复杂的。因此，通过阅读开源系统的源代码，能更好地掌握相关的内容。海翔的书能起到这个作用。

第三，我和海翔有师生之谊。本世纪初我们都在人大金仓工作，一同开发金仓数据库系统。海翔是公司员工，同时在中科大就读，我是他的企业导师。海翔平时话不多，少言寡语，但是心里有想法，是那种先做再说的性格。之后，由于种种原因，我离开了金仓回到学校任教。也就逐渐失去了和海翔的联系，直到有一天，他给我送来了他的一本书稿——《数据库查询优化器的艺术》，让我着实吃惊不小。我知道，在 IT 企业工作压力大，加班是常态。如果不是自己心里有一个目标，坚持不懈，





放弃许多休息时间，是很难写出有深度的著作的。我在心里为海翔点赞。之后我们又“失联”了，估摸着他是否又在做什么大事，果不其然，他又完成了自己的第二本著作，《数据库事务处理的艺术：事务管理与并发控制》。再次祝贺海翔！

写到这里，我也该停笔了。希望读者能从海翔的书中看到他对数据库事务处理的思考，以及他奋斗的影子。

杜小勇

中国计算机学会数据库专委会主任

出版社代注：

杜小勇教授，教育部科学技术委员会学部委员，中国计算机学会（CCF）常务理事、专委工委主任、数据库专委会主任，数据库领域著名专家，中国人民大学理工处处长。







## 推荐序二

数据库的基本特征之一是支持多用户共享数据,而事务管理和并发控制是提供这一支持的核心技术,是大型数据库有别于某些表格管理软件的关键特性之一,也是实现一个大型数据库管理系统时会面临的最有挑战性的技术之一。

本书从基本原理和案例系统分析深刻论述了该领域的进展现状与典型实现技术,特别是源码分析对有志于学习或定制开源数据库管理系统的开发人员提供了很好的参考。对数据库有关的各类从业者了解这一主题提供了难得的参考资料。

海翔热爱数据库研发,对数据库技术一直抱有一颗坚韧、执着之心,本书是他的经验和思索的体现,值得仔细研读。

张孝(博士)

中国人民大学信息学院副教授





## 推荐序三

事务是数据库的核心概念之一，提供 ACID（原子性、一致性、隔离性、持久性）特性的事务处理，是数据库系统能够商用化，并用来支持金融级业务的核心技术之一。一方面，大学本科或研究生的数据库教材更多的是偏向于事务的基本概念或基本技术的介绍；另一方面，现有大部分的技术文档对事务管理和并发控制的内容介绍缺乏重要的实现关键细节，无法有效地帮助学习者建立理论与实践之间的紧密联系。

而海翔的这本《数据库事务处理的艺术：事务管理与并发控制》：

- 不仅有对事务管理和并发控制的原理性介绍，如事务模型、基于封锁的并发控制技术、基于 MVCC 的并发控制技术等
- 还有对现有主流开源数据库，如 PostgreSQL、MySQL 的关于事务管理和并发控制的实现机制在源码级的深入剖析
- 再有多种商业数据库的实现机制分析，如 Oracle 和 Informix
- 更有新的技术的剖析，如 “write-snapshot isolation” “Serializable Snapshot Isolation (SSI)”
- 甚至还有理论、实现之间的优劣对比，以及多种实现技术的优劣对比，多角度地对比拓展了本书的纵深

这本书具备较好的深度、广度、新度，这让我十分期待。

我认识海翔已有十二载，在此期间，海翔一直从事于国产数据库的研发工作，取得了诸多令人瞩目的成绩，这本书是对他这些年理论探索和实践经验的总结，我相信对于志在从事数据库相关领域开发和研究的朋友，通过本书的学习，都能从中受益。

卢卫（博士后）

中国人民大学信息学院副教授

（发表多篇 SIGMOD、VLDB、ICDE 学术论文）





## 推荐序四

事务管理和并发控制是数据管理技术中最重要的几个主题之一，它保障了数据的一致性以及系统的性能。

本书聚焦于数据库中的事务处理，从原理、主流数据库实现、源码级实现三个角度进行了深度的探讨。尤其是后两个部分的介绍，让本书成为不可多得的有关 DBMS 事务管理模块内部技术细节的参考资料。

本书不仅对于从事各类数据库内核研发工作的人员具有很高的参考价值，对于从事数据库应用开发、DBA 乃至学习数据库的朋友们来说也值得一读。本书作者在数据库研发领域深耕多年，书中的字里行间显示出他扎实的实践经验和深厚的理论功底，相信各位读者都能从本书中受益匪浅。

彭煜玮（博士）

武汉大学计算机学院副教授 / 《PostgreSQL 数据库内核分析》作者

## 推荐序五

——庖丁解牛 高山仰止

海翔的新书挥就，单只书名就让我顿生敬意，事务处理是数据库技术中最为核心的部分，也是不同数据库之间暗暗较量的关键所在。以此为题的著作，我们熟知的就是 Jim Gray 的《事务处理概念与技术》，该书一直是数据库领域的筑基之作，令人高山仰止；现在海翔的著作以独到之角度阐释事务原理与并发控制，以庖丁解牛之刀为广大数据技术从业者剖析出宝贵的关节，实在是让人手不释卷的又一机缘。

在我熟悉的 Oracle 数据库领域，我一直认为其独到的事务处理机制，成就了 Oracle 数据库今天的地位。比如 Oracle 选择了提交读作为主要的事务隔离级别，进而通过 Undo 机制提供一致性读，而从 Oracle 6 引入的改变向量（Change Vector）技术更成为了 Redo 和 Undo 机制的核心，并由此保障数据库能够从一个一致性的状态不断迁移到下一个一致性状态，Oracle 数据库后续的很多卓越特性都由此构建，包括 Data Guard 技术、闪回技术等，延承至今。

而随着时代的进步，软硬件性能的提升，数据库产品也在不断进步，比如阿里巴巴的 OceanBase 基于内存短事务的设计，就不需要 Undo 日志，而 Oracle 公司最新透露的研发计划表明他们正在基于 NVRAM 进行内核改写，这其中最关键的就是对于 Redo 机制和事务的优化。

把握历史，兼顾当下，站在前人肩上才能跟上时代的步伐。海翔、冯毅和鹏程，正是站在这样的起点上。他们不仅精研数据库原理，更加是不同数据库内核的研发者，不同的从业经历，使得他们可以从横向和纵向对不同数据库进行剖析，Informix、Oracle、PostgreSQL 和 MySQL 等流行数据库的事务原理在书中均有呈现，真知灼见，俯仰可得。

这是一本理论结合研发，研发兼顾实践的实力之作，我相信也是作者们致敬大师的心血呈现，我期待这本书能够尽快呈现在案头，一睹为快！

盖国强

云和恩墨创始人 / Oracle ACE 总监

## 推荐序六

事务处理是数据库区别于其他系统的关键特质，JimGray 的《Transaction Processing》一书是打开我对数据库世界认识的一扇窗口。当海翔兄告诉我其在撰写《数据库事务处理的艺术》之书时很是感到震惊，因为这并不是一件容易的事情，同时也不是一件任何人都能干好的事情。

我相信本书的作者李海翔，我的好友，能将数据库事务处理的内部机制描述清楚。一方面，其从事数据库内核行业很多年，在这一方面有着非常深厚的内功基础。另一方面，其擅长于将复杂的问题通过简单的语句描述清楚。相信任何同学在看过本书之后都会有这样的感觉。更为难得的是，本书在事务处理这部分对比了当前主流的多种数据库实现，想必本书必将成为经典之作。

学习 MySQL 看姜老师的书，学习优化器和事务处理，就看海翔老师的作品吧。

姜承尧

腾讯金融支付数据库运营与研发部副总监 /

《MySQL 技术内幕》系列图书作者 /Oracle ACE



# 前言

## 前言

### 为什么写这本书

关系数据库管理系统有两大核心技术，一是事务处理，二是查询优化。对于数据库技术从业者来说，如果能把这两种技术珠联璧合融于一体，则如同长出两只翅膀，能助君振翅高飞、翱翔万里。

2017年，笔者的第二本著述即将问世。本书是一本讨论数据库事务管理和并发控制技术的书，融合了事务管理和并发控制原理、主流数据库的事务实现、开源数据库内核事务处理和并发控制源码剖析。本书能够帮助技术爱好者掌握事务处理的核心技术——事务管理和并发访问控制。

笔者期望，把自己对技术的深度探索以文字形式分享给读者，与数据库技术爱好者一起享受数据库内核技术之美。于是工作之余，周末枯坐敲击键盘，夜晚遐想苦思代码，便有了这本小作。

### 什么是艺术

书名冠以艺术一词，笔者实不敢当。艺术之大，笔者仰望如高山、如皓日。

惴惴之际，笔者试对艺术做一注解，谬误之词，请读者见谅。

艺术，美也；工程技术，美之实现也。数据库的工程实现，有美其中。代码有形之体美，算法有神之韵美，数据库内核之大之复杂故有极深的逻辑之美。爱美之心，笔者如同万千生灵，亦有之。于是因爱美而去尝试展示数据库之美，才有了笔者第一著述《数据库查询优化的艺术》和第二本著述《数据库事务处理的艺术》。

### 本书的主要内容

本书主要讨论的内容包含四个话题：

□话题一：事务处理技术中的一致性和隔离性，即ACID技术中的C和I。

□话题二：对于一致性，本书主要从事务模型和并发控制的角度出发进行讨论，着手于原理怎

样保障正确性、主流数据库怎样实现各种并发控制技术等探讨。比如封锁控制技术等。

□话题三：对于隔离性，本书以并发控制技术为出发点，在明确数据库的一致性之后，从数据库性能的角度出发讨论数据库隔离性的实现技术，比如著名的 MVCC 技术。

□话题四：从工程实践的角度，讨论 Informix、Oracle、PostgreSQL 和 MySQL 是怎样实现一致性和隔离性的。

本书分四篇，三个话题贯穿始终，分别从原理（第一篇）、主流数据库实现（第二篇）、源码实现（第三、四篇）三个角度进行了深度讨论。

原理是灵魂，是统领工程实现的纲，事务处理就是要保障数据的一致性。所以第一篇从数据异常现象这个引子出发，对并发控制技术的原理进行讲述，并分析了多种并发控制技术是如何解决数据异常现象的。这有别于经典的数据库理论书籍，本书不单纯为讲理论而讲原理，而是结合原理和工程实践中的知识和关键点讲原理，使得原理在工程实践中有落脚点和着力之处。

尽管原理相同，但是不同数据库的工程实现却不同。所以第二篇选取了主流的数据库系统，包括 Informix、Oracle、PostgreSQL 和 MySQL，对它们的一致性和隔离性的实现技术，以及工程实现的背后所折射的原理和思想进行了深度探索和对比。以期帮助读者对事务管理和并发控制技术的原理有进一步的认识，对主流数据库的实现情况有全面的掌握，更期望读者能把握主流数据库在工程设计方面的不同考量之处（基于相同原理却有着不同实现）。对于 Informix 和 Oracle，本书从宏观层面分析了其事务和并发访问控制技术，目的是从技术角度打开读者视野，探索事务管理和并发控制的实现技术，如 Informix 是纯粹的基于封锁的并发访问控制机制，我们可以品味到原汁原味的基于封锁的并发访问控制技术。

纵观全书，有理论和实践的对比、有不同实践方式的对比、还有新老技术的对比，等等，其妙处可在书中寻找并细加品味。仔细揣摩之后，您也许可以发现，在这个世界上，同是灿烂的生命却有着姹紫嫣红的不同。

另外，最能帮助掌握核心技术的，只有源码。本书的重点在于运用源码对原理进行剖析。所以，本书第三篇揭示了 PostgreSQL 事务处理的技术之美；第四篇揭示了 MySQL 和 InnoDB 事务处理的技术之美。而全书中，似在不经意间，在山石小径旁，长满了对原理、对工程实现的比较，以展示美与美的不同。如果读者能够在读书之际，自己再多做比较，则会领悟到更多的美……

## 本书的主要特色

从工程实践的角度看，本书通过原理分析、实例分析、代码分析，进行了三个对比：

□理论对比：对比事务处理中各种并发控制技术的异同，以扩展读者对事务处理技术理解的深度。并发控制的技术有很多，本书不为单纯讲每一个技术而罗列算法本身的内容，而是着眼

于对各种算法的差异进行对比,以揭示不同算法背后的思想。

□**实现对比**:对比四大主流数据库事务处理技术的异同,以扩展读者对事务处理技术理解的广度。本书不单纯讲述各主流数据库事务相关的 SQL 语法,而是着力于透视事务处理和并发控制技术实现方式之背后的设计考虑点,并随时对比各个主流数据库对于同一个问题的不同实现方式,以展示工程技术之美。

□**源码对比**:依托代码,直接分析和对比 PostgreSQL 和 MySQL(InnoDB)事务处理技术的异同,以帮助读者结合事务处理技术的原理和工程实践,真正理解和掌握数据库事务处理的技术。只有深入到源代码,才不会有雾里看花的感觉,置身于代码里,能美在其中、乐在其中。

## 资源及勘误

由于水平有限,书中难免会有差错和遗漏,希望广大读者能把发现的问题告诉我,将不胜感谢。本书的进步和完善,有您的帮助和支持,定能再上层楼。您可以发送邮件到:database\_XX@163.com 或 bluesea2db@gmail.com。由于时间有限,也许我不能答复所有的电子邮件,但是一定会定期整理并发布到我的博客([http://blog.163.com/li\\_hx](http://blog.163.com/li_hx))和 GitHub(<https://github.com/bluesea2DB/DB-MyBooks.git>)。

书中分析、探索的数据库源码下载地址如下:

PostgreSQL, V9.6.1, 源自: <http://www.postgresql.org/ftp/source/v9.6.1/>。

MySQL, V5.7.17, 源自: <http://dev.mysql.com/downloads/mysql/>。

## 致谢

本书承蒙教育部科学技术委员会学部委员、中国计算机学会(CCF)常务理事/数据库专委会主任、数据处理领域著名专家、中国人民大学杜小勇教授指导并作序。杜老师是我的恩师,杜老师待人文物、析事明理、治学严谨,一直是我心中的楷模,也是我前行路上的向导。十几年的熏陶,杜老师以人格之美、治学之严,熏陶了我,进而影响了我写书的风格。心怀感激,念念在心。

在我的生命中,家人是最重要的。再一次伏案疾书,不舍昼夜,亲人们再一次给予我强大的动力,让我内心充满暖意和歉意。特别是近一年来,我家小果果生命的孕育,亦陪伴、激励着我,在小果果来到这个世界感受生命之美时,努力把书写好、写美。特别感谢我的父母、妻子和果果,动力所在,有爱有美。十年的果果,因爱而感受到生命的美。

感谢为本书撰写推荐序的张孝博士、卢卫博士后、彭煜玮博士。他们是数据库学术界的青年模范,潜心学术钻研,同时还奋战在数据库和大数据研发一线,理论和实践结合,是推动国内数据库研发水平向前不断前进的重要动力。这种治学精神和实践行动,正是本书在精神层面所向望的:力求把



理论和实践结合起来，博学之，审问之，慎思之，明辨之，笃行之。

感谢为本书撰写推荐序的盖国强先生、姜承尧先生。他们都是数据库工业界的领跑者、实践者。他们不仅技术精湛，而且对数据库的理解、认知和熟悉程度，一直是我所敬佩的。盖国强先生、姜承尧先生著述颇多，是 Oracle、MySQL 等数据库技术的布道师。

感谢编辑杨福川先生和李艺女士为本书付出的努力和耗费的心血，书名源于杨先生，书稿样式改于李女士。谢谢他们，为本书注入心血。

感谢每一位读者，一起进步，你们将是我继续进步的新动力。

再一次感谢为开源社区无私奉献的人们，感谢经典的数据库著述和论文的作者们，本书成书的过程中，参考了相关的资料和信息，书中也一一标注。在此，一并感谢这些无私的奉献者们。没有他们，本书会逊色很多。

写下自己的第二本书，依旧惭愧本书尚不能达到书名标识的高度，依旧打算迈步从头越，依旧有着努力的心情，不一样的是书籍内容，依旧一样的是一颗忐忑的心，一颗期待的心，惴惴不安中期待本书对读者有所帮助，书有值，开卷有益。

李海翔

# 目 录

推荐序一
推荐序二
推荐序三
推荐序四
推荐序五
推荐序六
前言

## 第一篇 事务管理与并发控制 基础理论

### 第 1 章 数据库管理系统的事务原理···2

1.1 事务模型要解决的问题·····2
1.1.1 为什么需要事务处理机制···2
1.1.2 事务机制要处理的问题—— 事务故障、系统故障、介质 故障·····4
1.1.3 并发带来的问题——三种 常见的读数据异常现象·····4
1.1.4 并发带来的问题——写— 写并发操作引发的数据异常 现象·····8
1.1.5 语义约束引发的数据异常 现象·····9

1.1.6 其他的异常·····11
1.1.7 深入探讨三种读数据异常 现象·····13
1.2 事务处理技术的原理·····17
1.2.1 什么是事务·····17
1.2.2 事务的属性·····20
1.2.3 ACID 的实现技术·····24
1.3 事务的模型·····26
1.4 并发控制技术·····27
1.4.1 并发控制技术的实现 策略·····27
1.4.2 并发控制技术的实现 技术·····28
1.5 日志技术与恢复子系统·····31
1.6 本章小结·····32

### 第 2 章 深入理解事务管理和并发 控制技术·····33

2.1 在正确性和效率之间平衡·····33
2.1.1 隔离级别·····34
2.1.2 快照隔离·····36
2.1.3 理解可见性·····39
2.2 并发控制·····40
2.2.1 基于锁的并发控制方法·····42

2.2.2	基于时间戳的并发控制方法	47
2.2.3	基于有效性检查的并发控制方法	52
2.2.4	基于 MVCC 的并发控制方法	53
2.2.5	基于 MVCC 的可串行化快照隔离并发控制方法	56
2.2.6	再深入探讨三种读数据异常现象	60
2.3	并发控制技术的比较	62
2.3.1	并发控制技术整体比较	62
2.3.2	S2PL 和 SS2PL 的比较	64
2.3.3	事务属性与并发控制技术的关系	65
2.3.4	SCO 和 SS2PL 的比较	66
2.3.5	TO 和 SS2PL 的比较	67
2.4	深入探讨隔离级别	68
2.4.1	隔离级别与基于锁的并发控制方法	68
2.4.2	隔离级别与各种并发控制技术	69
2.5	事务的管理	70
2.5.1	事务的开始	71
2.5.2	事务的提交	71
2.5.3	事务的中止与回滚	72
2.5.4	子事务与 SAVEPOINT	72
2.5.5	长事务的管理	73
2.5.6	XA	74
2.6	事务相关的实战问题讨论	75
2.7	本章小结	76

## 第二篇 事务管理与并发控制应用实例研究

### 第 3 章 Informix 事务管理与并发控制

3.1	Informix 的事务操作	78
3.1.1	开始事务	78
3.1.2	提交事务	79
3.1.3	回滚事务	80
3.1.4	XA 事务	80
3.1.5	事务模型	82
3.2	Informix 的封锁技术	83
3.2.1	锁的级别	83
3.2.2	锁的粒度	84
3.3	隔离级别与数据异常	85
3.3.1	Informix 支持的隔离级别	85
3.3.2	隔离级别与日志的模式	86
3.3.3	写偏序异常	87
3.4	本章小结	88

### 第 4 章 PostgreSQL 事务管理与并发控制

4.1	PostgreSQL 事务操作	89
4.1.1	开始事务	90
4.1.2	提交事务	90
4.1.3	回滚事务	90
4.1.4	XA 事务	91
4.1.5	自动控制事务	91
4.2	SQL 操作与锁	92
4.2.1	锁的研究准备	92
4.2.2	INSERT 操作触发的锁	94
4.2.3	SELECT 操作触发的锁	94



4.2.4	SELECT FOR UPDATE 操作触发的锁	97	5.4	隔离级别与数据异常	131
4.2.5	UPDATE 操作触发的锁	100	5.4.1	SQL 标准定义的三种读 异常	131
4.2.6	DELETE 操作触发的锁	103	5.4.2	写偏序异常	134
4.2.7	ANALYZE 操作触发的锁	106	5.5	本章小结	138
4.2.8	CREATE INDEX 操作触 发的锁	106	<b>第 6 章 Oracle 事务管理与并发 控制</b>		139
4.2.9	CREATE TRIGGER 操作 触发的锁	107	6.1	Oracle 的事务操作	139
4.2.10	锁的相关参数	108	6.1.1	事务管理	139
4.3	隔离级别与数据异常	108	6.1.2	事务属性和隔离级别	140
4.3.1	SQL 标准定义的三种 读异常	108	6.1.3	XA 事务	141
4.3.2	写偏序异常	115	6.2	Oracle 的封锁技术	142
4.4	本章小结	118	6.2.1	元数据锁的级别	142
<b>第 5 章 InnoDB 事务管理与并发 控制</b>		119	6.2.2	用户数据锁的级别	143
5.1	InnoDB 的事务模型	119	6.3	MVCC 技术	145
5.1.1	开始事务	120	6.3.1	MVCC 的历史	145
5.1.2	提交事务与回滚事务	121	6.3.2	深入理解 MVCC	147
5.1.3	MySQL 的 XA	122	6.3.3	Oracle 的 MVCC	149
5.2	InnoDB 基于锁的并发控制	123	6.4	隔离级别与数据异常	157
5.2.1	基于封锁技术实现基本的 并发控制	123	6.4.1	Oracle 支持的隔离级别	157
5.2.2	锁的种类	124	6.4.2	写偏序异常	158
5.2.3	锁的施加规则	127	6.5	本章小结	160
5.2.4	获取 InnoDB 行锁争用 情况	129	<b>第三篇 PostgreSQL 事务管理 与并发控制源码分析</b>		
5.2.5	死锁	129	<b>第 7 章 PostgreSQL 事务系统的 实现</b>		162
5.3	InnoDB 基于 MVCC 的并发 控制	130	7.1	架构概述	162
			7.1.1	事务和并发控制相关的 文件	162

7.1.2	事务相关的整体架构	164
7.2	事务管理的基础	166
7.2.1	事务状态	166
7.2.2	事务体	171
7.2.3	事务运行的简略过程	172
7.3	事务操作	173
7.3.1	开始事务	173
7.3.2	事务提交	177
7.3.3	日志落盘	179
7.3.4	事务回滚	180
7.3.5	clog	185
7.4	子事务的管理	186
7.4.1	子事务与父事务的区别	186
7.4.2	保存点	187
7.5	本章小结	188

## 第 8 章 PostgreSQL 并发控制系统的实现——封锁

8.1	锁的概述	189
8.1.1	锁操作的本质	189
8.1.2	与锁相关的文件	190
8.1.3	与锁相关的内存初始化	191
8.2	系统锁	192
8.2.1	SpinLock	192
8.2.2	LWLock	198
8.2.3	SpinLock 与 LWLock 比较	213
8.3	事务锁	214
8.3.1	锁的基本信息	214
8.3.2	ReguarLock	221
8.3.3	行级锁	232
8.3.4	Advisory lock (劝告锁)	237
8.4	事务锁的管理	239

8.4.1	获取锁	239
8.4.2	锁查找或创建	242
8.4.3	释放锁	243
8.4.4	锁冲突检测	244
8.5	死锁检测	247
8.5.1	数据结构	247
8.5.2	等待获取锁与死锁处理	248
8.5.3	死锁检测	251
8.5.4	进程唤醒	252
8.6	从锁的角度看用法	254
8.6.1	AccessShareLock	254
8.6.2	RowShareLock	256
8.6.3	RowExclusiveLock	257
8.6.4	ExclusiveLock	258
8.6.5	其他的锁	260
8.7	本章小结	262

## 第 9 章 PostgreSQL 并发控制系统的实现——MVCC

9.1	快照	264
9.1.1	相关文件	264
9.1.2	数据结构	265
9.1.3	快照的类型	268
9.1.4	快照的管理	268
9.1.5	可串行化隔离级别的快照	271
9.2	可见性判断与多版本	273
9.2.1	可见性判断	273
9.2.2	多版本实现	282
9.3	可串行化快照原理	285
9.3.1	理论基础	285
9.3.2	算法实现	287

9.4 PostgreSQL 可串行化快照的实现 .....	289
9.4.1 PostgreSQL 的状况 .....	289
9.4.2 PostgreSQL 实现 SSI 的理论基础 .....	289
9.4.3 谓词锁数据结构 .....	297
9.4.4 谓词锁操作 .....	306
9.4.5 冲突检测 .....	321
9.5 隔离级别 .....	336
9.5.1 隔离级别 .....	336
9.5.2 各种隔离级别的实现 .....	337
9.6 本章小结 .....	340

## 第四篇 InnoDB 事务管理与并发控制源码分析

第 10 章 InnoDB 事务系统的实现 .....	342
10.1 架构概述 .....	342
10.1.1 事务和并发控制相关的文件 .....	342
10.1.2 事务相关的整体架构 .....	344
10.2 事务管理的基础 .....	346
10.2.1 事务状态 .....	346
10.2.2 表示事务的数据结构 .....	348
10.2.3 UNDO 日志与回滚 .....	349
10.2.4 REDO 日志 .....	350
10.2.5 内部事务的处理 .....	352
10.2.6 Mini-Transaction .....	352
10.3 事务操作 .....	353
10.3.1 InnoDB 的初始化 .....	354
10.3.2 开始事务 .....	354

10.3.3 提交事务 .....	359
10.3.4 日志落盘 .....	364
10.3.5 回滚事务 .....	367
10.3.6 Mini-Transaction 的提交 .....	371
10.3.7 Mini-Transaction 的回滚 .....	373
10.3.8 SAVEPOINT .....	373
10.3.9 XA .....	375
10.3.10 事务的其他内容 .....	375
10.4 InnoDB 事务模型 .....	378
10.5 本章小结 .....	382

## 第 11 章 InnoDB 并发控制系统的实现——两阶段锁 .....

11.1 锁的概述 .....	383
11.1.1 锁操作的本质 .....	383
11.1.2 全局锁表 .....	384
11.1.3 封锁系统的架构 .....	384
11.2 系统锁 .....	386
11.2.1 读写锁 .....	386
11.2.2 Mutex 锁 .....	394
11.2.3 其他锁 .....	401
11.3 事务锁之记录锁 .....	401
11.3.1 记录锁的基本数据结构 .....	402
11.3.2 记录锁 .....	408
11.3.3 记录锁与隔离级别 .....	423
11.4 事务锁之元数据锁 .....	433
11.4.1 元数据锁的数据结构 .....	433
11.4.2 元数据锁的管理与使用 .....	450
11.4.3 死锁处理 .....	468
11.5 SQL 语义定义锁 .....	476
11.5.1 锁的粒度 .....	476
11.5.2 重要的数据结构 .....	478

11.5.3 InnoDB 对接 MySQL Server.....	480	12.2 可见性判断 .....	506
11.6 其他类型的锁 .....	493	12.2.1 可见性原则 .....	506
11.6.1 Mini-Transaction 加锁 .....	493	12.2.2 二级索引的可见性 .....	509
11.6.2 事务锁之谓词锁 .....	494	12.3 多版本的实现 .....	509
11.7 事务与锁 .....	499	12.3.1 多版本结构 .....	509
11.8 本章小结 .....	500	12.3.2 多版本生成 .....	510
<b>第 12 章 InnoDB 并发控制系统</b>		12.3.3 多版本查找 .....	510
<b>的实现——MVCC .....</b>	<b>502</b>	12.3.4 多版本清理 .....	511
12.1 数据结构 .....	503	12.4 一致性读和半一致性读 .....	511
12.1.1 MVCC .....	503	12.4.1 一致性读 .....	512
12.1.2 Read View 快照 .....	504	12.4.2 半一致性读 .....	512
12.1.3 事务与快照 .....	505	12.5 本章小结 .....	513
		<b>附录 TDSQL 简介 .....</b>	<b>514</b>



## 第一篇

# 事务管理与并发控制基础理论

本篇介绍数据库管理系统的事务处理技术，从数据库的事务理论出发界定事务处理技术的范围，讨论了事务机制应对的问题、事务处理的理论基础和并发控制技术。

全篇立足于数据库事务处理的基本理论，第1章界定了本书讨论的事务处理技术所解决的问题，帮助读者理解事务处理技术的背景和需求。在笔者看来，理解问题比掌握解决问题的方式更为重要。

第2章讲述事务处理技术的基本概念和理论，但不是对经典教科书的翻录，而是结合笔者多年数据库内核开发的心得和经验，融入自己的理解，帮助读者从众多的头绪中理出线索。当然，要想彻底理解掌握事务处理的基础理论，还需要系统、全面地阅读经典的事务处理教材。

如果能够把阅读经典教材和本书相结合，并对比印证，则如步入百花丛中：百花丛中识花香，迷雾梦里寻捷径，这是一本好书所能做到的事情。

但愿书长久，十年共婵娟。

## 第 1 章

# 数据库管理系统的事务原理

数据库管理系统 (DataBase Management System, DBMS, 以下简称“数据库”), 是位于用户与操作系统之间的一层数据管理软件, 功能主要包括: 数据定义、数据操纵、数据库的运行管理、数据库的建立和维护<sup>⊖</sup>等。

数据库的事务处理机制, 是数据库技术的基石。掌握数据库技术, 必须要掌握事务处理的技术, 这样才能把握数据库的核心技术。同时, 也必须了解数据库为什么会需要事务处理技术, 即事务处理技术要解决的问题, 这一点, 就是我们在第一章第一节开门见山地提出的问题。

### 1.1 事务模型要解决的问题

事务模型, 是电子交易操作的保障。没有事务模型, 并发操作以及操作中途的系统异常将可能使数据发生混乱, 本节就各种潜在的问题进行讨论。

#### 1.1.1 为什么需要事务处理机制

数据库为什么需要事务处理机制?

面对这个问题, 一些数据库相关从业者掩卷思考, 给出以下几个答案:

答案一: 数据库的事务处理机制, 主要是解决脏读、不可重复读、幻读等问题的。

答案二: 数据库的事务处理机制, 主要是实现了 ACID 特性的。

答案三: 数据库的事务处理机制, 主要是以多版本两阶段封锁协议实现可串行化快照隔离。

---

⊖ 引自《数据库系统概论》王珊, 萨师喧著。

以上答案，哪个正确呢？其实，没有一个是正确的。

对于第一个回答，提出了事务处理技术中可能出现的三种经典问题，这三种问题，是数据库事务处理技术中并发控制模块为实现不同隔离级别而要解决的三个问题，不是事务处理机制面临的问题。所以第一个回答，以局部问题来覆盖整体，以偏概全了。

第二个回答，提出了 ACID 四大特性<sup>⊖</sup>，这四个特性在关系型数据库中一个也不可少，但这仅仅是“四个特性”，并不是我们需要数据库事务处理机制的根本原因。

第三个回答，提出了多项事务处理机制中需要的技术，如多版本（Multiversion concurrency control, MVCC）、两阶段封锁（2 Phase Lock）、快照隔离（Snapshot Isolation）等，但是这些技术仅仅是事务处理机制中的一些重要技术，用于并发控制管理和隔离级别的实现，并不是需要事务处理机制的原因。

那么，什么是数据库事务模型被提出的原因呢？

我们先回避问题，绕道看如下的一个实际用户操作，从 A 账户转账 50 元到 B 账户，其过程如表 1-1 所示。

表 1-1 事务处理机制的作用

第一种情况，没有事务处理机制	第二种情况，有事务处理机制
1. read(A) 2. A := A - 50 3. write(A) 4. read(B) 5. B := B + 50 6. write(B)	<b>begin</b> 1. read(A) 2. A := A - 50 3. write(A) 4. read(B) 5. B := B + 50 6. write(B) <b>commit</b>
假设第 3 行“3. write(A)”操作完成，A 账户已经少了 50 元	假设第 3 行“3. write(A)”操作完成，A 账户已经少了 50 元
从第四行开始，数据库宕机；重启后，A 账户少了 50 元但是 B 账户却没有增加 50 元	从第四行开始，数据库宕机；重启后，事务没有提交，A 账户尽管少了 50 元，但是事务因没有成功提交，导致 A 减少 50 元的操作“失效”
那么，A 少了的 50 元，哪里去了？	那么，A 少了 50 元的操作，被回滚掉了，A 不会有经济损失
如果发生这样的事情，谁还敢用数据库来记录交易信息？	如果发生这样的事情，A 是不是不用担心划账失败了？

事务处理机制，就是要保证用户的数据操作动作对数据是“安全的”。

那么，什么样的操作是安全的呢？数据只有在带有“ACID”四个特性的事务处理机制的保障下，才可以认为是“安全的”。

这里提到的大家耳熟能详的“ACID”四个特性，正是事务模型的核心内容。事务概念及其相关技术（为实现“ACID”四个特性的技术）的提出，给应用开发人员抽象出一个非常好用的数据处理模型，这个模型可以保证：操作间串行执行，执行中不必担心出错。这

⊖ ACID，A是原子性，C是一致性，I是隔离性，D是持久性。详见1.2.1节。

就是事务模型被提出的原因。A 保证了操作（一些有完整逻辑意义的数据读写动作）要么成功要么失败，A 和 C 保证了数据不会因写操作发生不一致，I 保证了在多会话并发读写同一份数据的情况下数据的完全一致（或数据可能不一致但尚可接受），D 保证了被修改的数据能长久地存储。

事务模型自被提出后，逐渐成为商业世界稳定有序运作的基石，数据和数据承载的交易事件的结果不会因系统故障被损伤。关系型数据库管理系统实现了事务模型，使得数据库在电子交易中发挥了非常重要的作用。这也是数据库为什么需要事务处理机制，即事务模型的原因。

### 1.1.2 事务机制要处理的问题——事务故障、系统故障、介质故障

事务模型是商业世界运作的基石，主要体现在交易处理电子化。作为一个软件系统，数据库系统会遭遇一些故障，如我们常说的事务故障、系统故障、介质故障等。

对于数据库系统，因为数据实在太为重要，数据库系统应该能够保证在出现故障时，数据依然满足 ACID 特性。

对于事务内部的故障，一般可分为预期的和非预期的。预期的事务内部故障是指可以通过数据库的事务处理机制发现的事务内部故障，这时数据库的事务处理机制提供了回滚操作保证了数据免受损害；非预期的事务内部故障如运算溢出故障、并发事务死锁故障、违反了某些完整性约束而导致的故障等，数据库系统依然可以通过回滚操作保证数据免受损害，所以回滚操作在事务处理机制中占有重要地位，不同的数据系统对回滚的实现方式也不尽相同。

对于系统故障，如数据库在运行过程中，由于硬件故障、数据库软件及操作系统的漏洞、突然停电等，数据库系统停止运行，所有正在运行的事务以非正常方式终止，需要数据库系统重新启动，这时，数据库系统为保障 ACID 特性，提供了基于 REDO/UNDO 的恢复机制，可以正确恢复到系统崩溃前的状态。

介质故障也称为硬故障，主要指数据库在运行过程中，由于磁头碰撞、磁盘损坏、强磁干扰、天灾人祸等情况，使得数据库中的数据部分或全部丢失。解决这一类故障，要依赖于备份系统和归档日志，归档的日志、系统运行期间记载的 REDO/UNDO 日志等日志相关内容，也是事务处理机制的一部分，日志管理部分的内容较多，一般的书籍会将日志管理独立为若干章节进行讨论。

总的来说，事务处理机制为应对这三类故障，提供了很多技术，如日志与恢复子系统、并发控制子系统、存储子系统等，和事务处理机制密切相关，我们将在 1.2 节继续进行讨论。

### 1.1.3 并发带来的问题——三种常见的读数据异常现象

数据不在 ACID 特性的保护下会发生不一致的现象，那么：



在 ACID 的保护下，是不是数据就一定不会产生不一致的现象呢？

在关系数据库系统中，多个会话（session）可以访问同一个数据库的同一个表的同一行数据。这样，对于数据而言，就意味着在同一个时间段内，有多个会话可以对其施加操作（或读操作或写操作），读写操作施加的顺序不同以及事务中 A 特性对事务结果的影响（或成功或失败，即要么提交要么中止），这三种因素叠加在一起，会存在几种对数据有不同影响的情况：

- **读－读操作**：如果同时只存在多个读操作，对于数据自身则没有影响；既读和读操作互不影响数据的一致性，读读操作可以并发执行。
- **读－写操作**：如果读写操作都存在，因写在前读在后（如脏读现象）、读在前写在后（如不可重复读现象），或者读在前写在后然后又读（如幻象现象），就可能因数据被写而导致另外一个读操作的会话读到错误的数据库。这个操作可以根据动作发生的先后顺序被细分为读－写操作、写－读操作。
- **写－写操作**：如果同时存在多个写操作，写－写操作直接改变了数据在同一时刻的语义，这就更不被允许，所以写－写操作通常不允许被并发执行。但是，如果不做并发控制，写－写并发操作也会带来数据异常现象（1.1.4 节探讨写－写操作引发的异常）。

这三种情况的第二种，对应了 SQL 标准中定义的三种数据异常的现象，注意这三种异常都是针对某个事务（第 2.2.1 节称这样的事务为“主事务”）的读操作而言的。SQL2003 标准对于数据异常现象的定义如下：

```
The isolation level specifies the kind of phenomena that can occur during the
execution of concurrent SQL-transactions.
The following phenomena are possible:
1) P1 ( "Dirty read" ): SQL-transaction T1 modifies a row. SQL-transaction T2
then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK,
T2 will have read a row that was never committed and that may thus be considered
to have never existed.
2) P2 ( "Non-repeatable read" ): SQL-transaction T1 reads a row. SQL-transaction
T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts
to reread the row, it may receive the modified value or discover that the row
has been deleted.
3) P3 ( "Phantom" ): SQL-transaction T1 reads the set of rows N that satisfy some
<search condition>. SQL-transaction T2 then executes SQL-statements that
generate one or more rows that satisfy the <search condition> used by
SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same
<search condition>, it obtains a different collection of rows.
```

怎么理解上面的内容呢？从 SQL 标准的定义可以看出：

- **首先**，涉及了并发事务（concurrent SQL-transactions）：至少有两个事务同时发生，如上面所举的三种异常现象中，分别使用了事务 T1 和 T2。但是，更为复杂的情况，

如下一节所谈到的“写偏序 (write skew)”现象的发生可以是两个或三个事务同时发生。

- 其次，涉及了隔离级别 (isolation level)：如果多个事务是“可串行化”的，则意味着事务之间不应该互相影响（不是“一定互不影响”而是“不应该互相影响”），即逻辑上不存在“读写”冲突，所以数据库引擎实现时应该避免“读写”冲突造成的数据不一致的现象。避免的含义就是在数据库引擎编码实现的时候，采取并发控制技术，消除“脏读”、“不可重复读”、“幻象”这三种现象。采取的手段是分级控制（不同的隔离级别），分别使用“已提交读 (READ COMMITTED)”、“可重复读 (REPEATABLE READ)”和“可串行化 (SERIALIZABLE)”这几种隔离级别，来应对这三种数据不一致的现象。
- 数据操作的对象，脏读和不可重复读是以“row”（一行）为单位，而幻象现象操作的是一个数据集（零行到多行）。

下面，我们把 SQL 标准的话语转为一个表格（如表 1-2 所示），以更好地理解一下三种异常现象。

表 1-2 SQL 标准定义的三种读数据异常现象

时间	脏读 P1 ( “Dirty read” )		不可重复读 P2 ( “Non-repeatable read” )		幻象 / 幻读 P3 ( “Phantom” )	
	T1	T2	T1	T2	T1	T2
t0	W(row)-Update		R(row)		R(rows)- WHERE<condition>	
t1		R(row)		W(row)-Update/Delete		W(rows)- Insert/Update =><condition>
t2	Abort			Commit	R(rows)- WHERE<condition>	
t3			R(row)			

说明：

- 表格头两行，表明 SQL 标准定义的三种异常现象，分别是脏读、不可重复读、幻象。
- 表格第一列，时间值列，表明时间值在逐渐增长，即  $t_0 < t_1 < t_2 < t_3$ 。
- 对于每一种异常现象，都分为 2 个列，分别是两个并发的事务，各自命名为 T1 事务和 T2 事务。
- 表格中的 R(row)，表示读 row 数据对象；W(row) 表示写 row 对象，读写操作的是同一个数据对象 row；W(row)-Update/Delete 后面的 Update/Delete 表示的写操作是 DML 语句中的 UPDATE 或 DELETE 语句产生的写数据库对象 row 的操作。
- 带有“R(rows)-WHERE<condition>”的表示 SQL 语句读数据（通常是 SELECT 语句）带有 WHERE 子句，此子句的条件表达式是“<condition>”。



- 带有 “W(rows)-Update/Insert-<condition>” 的表示写操作生成的数据满足与其他事务读操作带有的相同的条件表达式 “<condition>”，且此写操作要么是 UPDATE，要么是 INSERT 语句。
- 带有阴影背景的，表示其对应时刻，如果发生对应的操作，将引发异常现象。
  - 脏读现象：按照时间顺序，T1 事务在 t0 时刻对 row 进行了修改（更新），T2 事务在 t1 时刻先读取了被 T1 修改了的 row 的值，但是 T1 在 t2 时刻中止使得对 row 的修改失效。如果数据库引擎不支持因并发操作避免数据异常，则 T2 在 t1 时刻读到的就是 T1 修改后的数据，但是这个数据在现实世界中不存在，对于事务 T2 而言，读取了被回滚掉的数据，即事务 T2 发生了脏读异常现象。另外，对于脏读现象，还存在如表 1-3 的两种变形情况。
  - 不可重复读现象：事务 T1 在 t0 时刻先读取 row 的旧值，事务 T2 在 t1 时刻对 row 进行了修改（更新或删除）然后提交事务使得修改生效，此时 row 因更新变为了新值或因删除而不再存在。接下来，事务 T1 在 t3 时刻再次读取 row 对象的值但是 row 的值已经是新值或者不存在了。对于事务 T1 而言，物是（同样是读 row 这个对象）人非（值已经和 t0 时刻读到的值不同），事务 T1 发生了不可重复读异常现象。
  - 幻象现象：事务 T1 在 t0 时刻带有特定条件地读取了 row 对象的数据，事务 T2 在 t1 时刻插入新的数据或更新其他旧数据但满足事务 T1 的特定 WHERE 条件，新的数据满足与事务 T1 同样的条件，当事务 T1 在 t3 时刻再次以同样的条件读取数据的时候，rows 对象的值已经有新加入的行（因插入而比第一次读多出了数据）。对于事务 T1 而言，物是（同样是读满足 “<condition>” 条件的多个 row 对象）人非（值已经和 t0 时刻读到的值不同），事务 T1 发生了幻象异常现象。幻象又被称为幻读，即第二次读操作读取了第一次读操作没有读到的 rows（一行或多行）。脏读异常的变形如表 1-3 所示。

表 1-3 脏读异常的变形

时间	脏读 P1 ( “Dirty read” )		撤销读 Aborted read		中间读 Intermediate read	
	T1	T2	T1	T2	T1	T2
t0	W(row)-Update		W(row)-Update		W(row)-Update	
t1		R(row)		R(row)		R(row)
t2	Abort			Commit		Commit
t3			Abort		W(row)-Update	
	事务 T2 读了一个将被撤销的事务 T1 写的数据		已经提交的事务 T2 读了一个将被撤销的事务 T1 写的数据		已经提交的事务 T2 读了事务 T1 临时写的一个中间状态的数据	

在了解了 SQL 标准定义的三种异常现象后，回到我们在本节开始提出的问题：  
在 ACID 的保护下，是不是数据就一定不会产生不一致的现象呢？  
答案已经很明确：即使数据库系统提供 ACID，除非我们使用“可串行化

(SERIALIZABLE)”隔离级别，否则数据在其他不同的隔离级别下还会产生数据不一致的现象。

有关这三种数据异常现象的更多探讨，请参见 1.1.7 节。

1.1.4 并发带来的问题——写 - 写并发操作引发的数据异常现象

上一节，我们探讨了三种读数据异常现象，请注意，异常现象发生在一个事务中后面的 READ 读操作上。这三种读数据异常现象被 SQL 标准定义。那么：

是不是对数据的并发操作只会产生上述的三种读数据异常现象？

事务概念的奠基人，Jim Grey 先生，在其著作《事务处理：概念与技术》中提到了一个“Lost Update”异常的概念，字面含义是“更新丢失”，这是一个写操作的异常，与上一节提到的三种读操作异常不同。

除了“Lost Update”异常外，这一节我们还将探讨另外一种异常，如表 1-4 所示。

表 1-4 写数据异常现象

时间	脏写		丢失更新	
	Dirty write		Lost Update	
	T1	T2	T1	T2
t0	W(row)-Update		R(row)	
t1		W(row)-Update		W(row)-Update
t2		Commit	W(row)-Update	
t3	Abort		Commit	

说明：

- 表格头两行，表明写写并发操作引发的两种异常现象，分别是脏写、丢失更新。
- 表格第一列，时间值列，表明时间值在逐渐增长，即  $t_0 < t_1 < t_2 < t_3$ 。
- 对于每一种异常现象，都分为 2 个列，分别是两个并发的任务，各自命名为 T1 任务和 T2 任务。
- 表格中的 R(row)，表示读 row 数据对象；W(row) 表示写 row 对象，读写操作的是同一个数据对象 row；W(row)-Update 后面的 Update 表示的写操作是 DML 语句中的 UPDATE 语句产生的写数据库对象 row 的操作。
- 带有阴影背景的，表示其对应时刻，如果发生对应的操作，将引发异常现象。
  - 脏写现象：按照时间顺序，任务 T1 在  $t_0$  时刻对 row 进行了修改（更新），任务 T2 在  $t_1$  时刻对 row 进行了修改（更新），如果没有并发控制，T2 对 row 的修改会生成新值，但是 T1 在  $t_3$  时刻回滚使得 T2 对 row 的修改失效，而 T1 的语义是：T1 自身对 row 的修改失效，这就把 T2 修改的值回滚掉。对于任务 T1 而言，回滚掉了不是自己修改的数据，即任务 T1 上发生了脏写现象。
  - 丢失更新现象：按照时间顺序，任务 T2 在  $t_1$  时刻对 row 进行了修改（更新），任务



T1 在  $t_2$  时刻对 row 进行了修改 (更新), 如果没有并发控制, T1 对 row 的修改会生成新值, 但是 T1 在  $t_3$  时刻提交使得 T2 对 row 的修改失效。对于事务 T1 而言, 覆盖掉了不是自己修改的数据, 即事务 T1 上引发了丢失更新现象 ( $t_3$  时刻如果是事务 T2 提交而不是事务 T1 提交, 也是丢失更新, 只是事务 T2 上引发了丢失更新现象)。

不管是读异常还是写异常, 并发控制技术都要规避这些异常, 保证数据在不同隔离级别下一致性不被破坏。

### 1.1.5 语义约束引发的数据异常现象

如果说数据在 ACID 特性 (带有了并发控制技术) 的保护下会发生不一致的现象, 那么:

在 ACID 和快照隔离级别技术 (多版本) 的保护下, 是不是数据就一定不再会产生不一致的现象呢?

答案是否定的。数据库系统中数据的异常, 在多种并发控制技术中已经被解决, 但这并不表明所有的异常都已经被解决, 更不表明不再有新的异常被发现。

我们知道, 数据库并发控制技术中有一个大名鼎鼎的技术, 称为快照隔离 (Snapshot Isolation)<sup>①</sup>, 这项技术解决了读和写之间的冲突, 在保证数据不会产生前面两节提到的读异常和写异常的情况下, 使得读写互不阻塞 (两阶段封锁技术中读写操作互相阻塞), 提高了并发度。

注意我们这里谈到的多版本是 “multi-version, 简称 MV”, 其相对于 “single-valued, 简称 SV”, 这个多版本是指并发控制技术中的数据项有多个版本, 其含义仅此而已。快照隔离并发技术是多版本并发控制 (Multiversion concurrency control, MVCC) 技术的一部分, 其发生作用, 需基于 “数据项存在多个版本”。

快照隔离并发控制技术的缺点, 是并不能真正保证事务为 “可串行化的”, 即事务间的并发操作依旧会引发数据异常现象, 但是这里的数据异常现象有别于前面提到的各种异常现象, 其异常现象是 “业务的逻辑语义” 引发的, 即除了抽象的读写操作, 数据间还应该满足一定语义, 即约束 (constraint)。

在快照隔离并发控制技术中并发的事务因不满足约束而发生的异常, 称为 “写偏序 (Write Skew)”, 这样的异常有两种, 参见表 1-5。

说明:

- 表格头两行, 表明写偏序异常现象的两种情况, 分别是由两个事务引发异常、三个事务引发异常。
- 表格第一列, 时间值列, 表明时间值在逐渐增长, 即  $t_0 < t_1 < t_2 < t_3 < t_4 < t_5 < t_6 < t_7$ 。
- 对于每一种异常现象, 都分为 2 个列, 分别是两个并发的事务, 各自命名为 T1 事务和 T2 事务。

① 详情参见 2.1.2 节。

表 1-5 写偏序异常的情况<sup>⊖</sup>

时间	两个事务写偏序		三个事务写偏序		
	T1	T2	T1	T2	T3
t0	x ← SELECTCOUNT(*) FROM doctors WHERE on-call = true			x ← SELECT current_batch	
t1		x ← SELECTCOUNT(*) FROM doctors WHERE on?call = true			INCREMENT current_batch
t2	IF x ≥ 2 THEN UPDATE doctors SET on-call = false WHERE name = Alice				Commit
t3		IF x ≥ 2 THEN UPDATE doctors SET on-call = false WHERE name = Bob	x ← SELECT current_batch		
t4	Commit		SELECT SUM(amount) FROM receipts WHERE batch = x-1		
t5		Commit	Commit		
t6				INSERT INTO receipts VALUES(x, somedata)	
t7				Commit	

□ 二个事务引发的异常现象（简单写偏序，Simple Write Skew）：按照时间顺序，T1 事务在 t0 时刻读取了在打电话的值班医生个数，T2 事务在 t1 时刻也读取了在打电话的值班医生个数。事务 T1 在 t2 时刻进行判断：如果在打电话的值班医生个数大于等于 2 人则请 Alice 停止打电话。事务 T2 在 t3 时刻进行判断：如果在打电话的值班医生个数大于等于 2 人则请 Bob 停止打电话。然后事务 T1 和 T2 分别提交。如果在这种并发的情况下，允许事务 T1 和 T2 都提交成功，则 t6 时刻，Alice 和 Bob 都停止了打电话。如果串行执行事务，先执行事务 T1 后执行事务 T2，Alice 会停止打电话但 Bob 不会停止，这与前一种情况的结果不同；如果先执行事务 T2 后执行事务 T1，Bob 会停止打电话但 Alice 不会停止，这与前一种情况的结果也不同；这表明前一种并发执行是非序列化的，而此时，事务 T1、T2 并发时违反了约束（约束为：如果同时打电话的人数大于等于 2 人则请 Alice 或 Bob 其中一个人停止打电话直到同时打电话的人数少于 2 人）发生了写偏序异常现象。对于简单写偏序，可以用一个形象化的图表示，如图 1-1 所示。

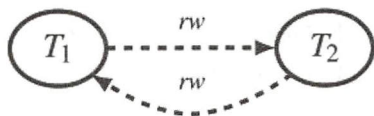


图 1-1 两个事务引发的异常现象优先图

⊖ 示例源自论文：Dan R. K. Ports, Kevin Grittner, Serializable Snapshot Isolation in PostgreSQL

□ 三个事务引发的异常现象 (Batch Processing)：对于这种情况，后两个并发更新事务 T3 和 T2 是可串行化的且不存在任何异常，但是一个只读事务 T1 出现在某个时刻却可能正好造成问题。所出现的问题是这样的，当事务 T3 提交时，T2 处于活跃状态，这时，事务 T1 启动要读取事务 T2 和 T3 涉及的数据 (current\_batch 和 receipts)，这时，事务 T1 的快照包括了事务 T3 的插入后的结果 (因为 T3 已经提交)；但是，事务 T2 没有提交，它的插入操作数据不包含在事务 T1 的快照中。在优先图 (如图 1-2 所示) 中会造成一个环 (有关如何形成这样的环，参见 2.2.5 节)，说明这样的调度是非可串行化的。

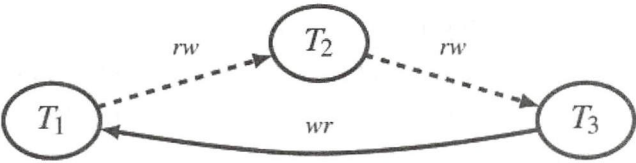



图 1-2 三个事务引发的异常现象优先图<sup>⊖</sup>

本节所述的这两种情况，如果使用优先图表示，都可以在参与操作的事务之间，画出一个环，存在环说明：调度是非可串行化的。为解决这样的问题，要求数据库引擎必须在事务提交时 (甚至是环一形成即立刻回滚其中的一个事务) 而不是在快照上检查完整性约束，以避免本节所述的不一致现象。

 更多的写偏序异常示例，可以参见：[https://wiki.postgresql.org/wiki/SSI#Read\\_Only\\_Transactions](https://wiki.postgresql.org/wiki/SSI#Read_Only_Transactions)。

1.1.6 其他的异常

在《A Critique of ANSI SQL Isolation Levels》这篇论文中，除了上面提到的几种异常现象外，还提到了另外两种异常，如表 1-6 和表 1-7 所示。

表 1-6 读偏序

	读偏序	
	Read Skew	
时间	T1	T2
t0	R(x)	
t1		W(x)、W(y)
t2		Commit
t3	R(y)	

⊖ 图1-1、图1-2的具体含义，请参见2.2.5节。



说明：

- 表格头两行，表明读偏序异常现象，是由两个事务引发异常。
- 表格第一列，时间值列，表明时间值在逐渐增长，即  $t_0 < t_1 < t_2 < t_3$ 。
- 对于读偏序异常现象，都分为 2 个列，分别是两个并发的任务，各自命名为 T1 事务和 T2 事务。
- 事务 T1 在  $t_0$  时刻读出数据  $x$ ，事务 T2 在  $t_1$  时刻对数据  $x$  和  $y$  进行了修改在  $t_2$  时刻提交，事务 T1 在  $t_3$  时刻读取  $y$ ， $y$  是被事务 T2 修改后的数据，此时已经不是  $t_0$  时刻事务 T1 读取  $x$  时对应的  $y$  值，数据形成了不一致状态（注意此时不是数据  $x$  处于不一致，而是  $y$  处于不一致）。

游标（Cursor），是数据库引擎提供的一种读取数据的方式。在数据库中，为了防止使用游标时其他事务并发修改游标所包括的数据，定义了游标稳定性（Cursor Stability）隔离级别，这样的隔离级别，是在读取当前数据项的时刻，在数据项上加锁，当游标从当前数据项移走则解锁，此后曾经被读取过的数据项不再被并发保护（尽管使用游标的事务没有提交或中止，还处于活动状态）。但是，在使用游标的时候，也会发生写－写异常，如表 1-7 所示。

表 1-7 游标丢失更新

时间	游标丢失更新	
	Cursor Lost Update	
	T1	T2
$t_0$	R(row)	
$t_1$		W(row)-Update
$t_2$	W(row)-Update	
$t_3$	Commit	

说明：

- 表格头两行，表明写－写并发操作引发的异常现象，游标丢失更新。
- 表格第一列，时间值列，表明时间值在逐渐增长，即  $t_0 < t_1 < t_2 < t_3$ 。
- 对于每一种异常现象，都分为 2 个列，分别是两个并发的任务，各自命名为 T1 事务和 T2 事务。
- 游标丢失更新现象：按照时间顺序，事务 T1 在  $t_0$  时刻读取 row 的值后，即释放了 row 上的锁使得 row 没有被并发保护，事务 T2 在  $t_1$  时刻对 row 进行了修改（更新），事务 T1 在  $t_2$  时刻对 row 进行了修改（更新），如果没有并发控制，T1 对 row 的修改会生成新值，但是 T1 在  $t_4$  时刻提交使得 T2 对 row 的修改失效。对于事务 T1 而言，覆盖掉了不是自己修改的数据，即事务 T1 上引发了丢失更新现象。这样的现象，本质上就是丢失更新，只是发生在了游标上，所以称为游标丢失更新。

我们在 1.1.3 至 1.1.6 节，介绍了各种因并发引发的异常现象，有多种技术（2.2 节介绍多种并发控制技术）可以解决这些异常现象，如下我们借用《A Critique of ANSI SQL





Isolation Levels》这篇论文中的一张表，来简单总结一下前面几节所谈到的各种异常，如表 1-8 所示。与隔离级别相关的内容，参见 2.1 节。

表 1-8 所有的异常和隔离级别的关系

Isolation Types Characterized by Possible Anomalies Allowed.								
Isolation level	P0 Dirty Write	P1 Dirty Read	P4C Cursor Lost Update	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	A5A Read Skew	A5B Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

说明：

- ❑ P0 Dirty Write：脏写，1.1.4 节表 1-3。
- ❑ P1 Dirty Read：脏读，1.1.3 节表 1-2。
- ❑ P4C Cursor Lost Update：游标丢失更新，1.1.6 节表 1-6。
- ❑ P4Lost Update：丢失更新，1.1.3 节表 1-2。
- ❑ P2Fuzzy Read：模糊读，即不一致读，1.1.3 节表 1-2。
- ❑ P3 Phantom：幻象，即幻读，1.1.3 节表 1-2。
- ❑ A5A Read Skew：读偏序，1.1.6 节表 1-5。
- ❑ A5B Write Skew：写偏序，1.1.5 节表 1-4。
- ❑ 与隔离级别相关的内容，参见 2.1 节。

### 1.1.7 深入探讨三种读数据异常现象

在 1.1.3 节，我们提出了并发带来的三种异常现象，分别是脏读、不可重复读、幻象，其中幻象又被称为“幻读”。

这一节，我们将以问答的形式，深入探讨与这三种异常现象相关的六个问题。

Q1：异常现象是发生在表 1-2 中的事务 T1 还是事务 T2？

换一个提问方式，我们都知道数据库提供了四种隔离级别，那么当三种异常发生的时候，是应该在事务 T1 中还是事务 T2 中设置隔离级别以避免数据异常现象的产生？

答：

首先，需要区分操作发生的主体。数据库启动多个会话，要想会话之间互不影响，则



需要标识哪个是当前会话、哪个是其他会话，然后设置隔离级别。有人认为，这个很简单，每个会话对于其自身就是当前会话。但是，对于数据库引擎而言，他是不知道哪个会话是当前的会话。

其次，需要考虑并发度。对于数据库引擎而言，如果因为数据库引擎不知道哪个会话是当前的会话，所以给每个会话的事务都设置不同的隔离级别，减少各个会话的事务互相影响，则会降低并发度。

所以，在数据库保留一个默认的隔离级别（通常是较低的隔离级别以提高并发度）给每个会话的事务后，如果需要高的隔离级别，就需要在自己的事务中设置想要的隔离级别，以使得本事务不受其他事务的影响。因此，在数据库系统中，我们要想使得事务 T1 不受其他事务的影响，则需要在事务 T1 内部设置隔离级别，使得本事务 T1 不受其他并发事务对事务 T1 要读写的数据造成影响。换句话说，就是使得本事务不发生异常现象。回看表 1-2 以及对表 1-2 的说明，我们可以看到如下内容：

□ 脏读现象：……事务 T2 发生了脏读异常现象。

□ 不可重复读现象：……事务 T1 发生了不可重复读异常现象。

□ 幻象现象：……事务 T1 发生了幻象异常现象。

这个结论看起来有点儿不“美”，即有点不协调，脏读是发生在了事务 T2，不可重复读和幻象是发生在了事务 T1。

如果数据异常都发生在事务 T1，则我们可以把事务 T1 想象成为“本事务”，事务 T2 是其他并发的任务，而我们当前可以操作的（即有权限可以设置隔离级别）只能是“本事务”，其他并发发生的事务也许是其他权限用户建立的会话，我们根本没有权限操作即没有权限去设置他人的隔离级别。

所以，数据异常现象，一定是发生在本事务当中的。我们统一以事务 T1 为本事务（也称为“主事务”），观察本事务和其他并发事务之间因读写操作的先后次序不同而造成的不同的数据异常。因此，我们修正表 1-2 的内容如表 1-9 所示。

表 1-9 修正后的 SQL 标准定义的三种读数据异常现象

时间	脏读 P1(“Dirty read”)		不可重复读 P2(“Non-repeatable read”)		幻象 P3(“Phantom”)	
	T1 主事务	T2	T1 主事务	T2	T1 主事务	T2
t0		W(row)-Update	R(row)		R(rows)-WHERE <condition>	
t1	R(row)			W(row)-Update /Delete		W(rows)-Insert/Update =><condition> ⊖
t2		Abort		Commit	R(rows)-WHERE <condition>	
t3			R(row)			

⊖ “W(rows)-Update/Insert =><condition>” 表示更新和插入操作的行满足“<condition>”。



对于 Q1 问题的回答, 依据表 1-9, 数据异常现象发生在表 1-9 中的主事务 T1 中。其实, 如上修改是为了明确并发事务的数据异常现象发生在哪里。即研究别的事务对于本事务的影响, 而不是讨论本事务对别的事务的影响。

对于数据库引擎的编码实现, 就是在一个会话中考虑本会话(主事务对应的会话)因不同的隔离级别(在主事务中设置的隔离级别)在发生冲突后而产生什么样的动作(可继续执行本事务内的后续 SQL 还是直接回滚掉本事务)。所以, 请注意, 动作发生的主体, 一定是主事务。而且, 异常是发生在主事务的读操作这样的动作上(指三种读异常)。

Q2: 从表 1-9 中看, 对于脏读现象, 写操作是事务 T2 执行 UPDATE 引发的, 那么, 事务 T2 的写操作可以是删除 (DELETE) 或插入 (INSERT) 吗?

答:

脏读, 强调的是主事务读取了一个不存在(因回滚而不存在)的数据。

如果事务 T2 的写操作是删除操作, 在 row 这行数据被删除后, 事务 T1 不可能在后面的时间点 t1 上读到同一个 row (SQL 标准中特意强调了 “that row”, 参见 1.1.3 节); 因此脏读现象中 “事务 T1 一定可以读到同一个 row” 是不能满足的, 因此, 脏读中的事务 T2 的写操作不可能是删除。

如果事务 T2 的写操作是插入操作, 在 row 这行数据被插入后, 事务 T1 能在后面的时间点上读到同一个 row (SQL 标准中特意强调了 “that row”); 因此脏读现象中事务 T1 一定可以读到同一个 row 的条件是被满足的, 因此, 脏读中的事务 T2 的写操作可以是插入。

Q3: 对于不可重复读现象, 事务 T2 的写操作是否可以插入操作?

答:

对于事务 T2 的 W(row) 操作, row 是一个已经存在的行, 这表明是在一个存在的特定对象上进行的操作, 所以不可能是插入操作。

Q4: 不可重复读和幻象有什么区别?

答:

首先, 这两种异常, 对于主事务 T1 而言, 都是先读取了数据, 之后因事务 T2 “写”了数据而事务 T1 再次读取数据的时候, 发生了异常。但是, 如表 1-9 所示, 不可重复读对于事务 T1 读取的是一个存在的确定的一行数据(意味着这行数据是存在的数据), 这个行数据本身被事务 T2 使用更新或删除操作而改变; 而幻象对于事务 T1 读取的是满足条件 “<condition>” 的多行数据(意味着 “<condition>” 是一个范围查找, 结果集不确定), 所以从第一次读取数据的操作的角度看, 前者是读取特定的行, 后者读取的是多行但可能不确定。不过, 对于所读的数据而言, 由此可以产生一个新的问题, 参见 Q5。

其次, 这两种异常, 对于事务 T2 而言, 都是 “写” 数据, 但是写操作的具体动作不同。如表 1-9 所示, 不可重复读对于事务 T2 的写操作是更新或删除操作, 而幻象对于事务 T2 的写操作是插入(插入的新数据满足条件)或更新(使不满足条件的数据在更新后满足条件)操作。

第三, 不可重复读和幻象最大的区别就是前者只需要 “锁住”(考虑)已经读过的数据,





而幻象需要对“还不存在的数据”做出预防。

Q5：对于幻象现象，事务 T2 的写操作是否可以更新操作或者删除操作？

答：

对于幻象现象中事务 T2 的 W(rows) 操作，如果操作是一个更新或删除操作，则表明这样的操作等同于（相似但不完全一样，区别参见 Q6）不可重复读（即是在多个行数据上进行更新或删除，即在多个行数据上批量化重演了不可重复读现象）。

如表 1-10 所示，比较不可重复读和幻象现象。如果我们认为幻象现象中的事务 T2 可以是更新或删除操作，则幻象就等同于不可重复读。实际上，幻象是不可重复读的一个特例，对于不可重复读现象，可以扩展其“R(row)/W(row)”的概念为“R(rows)/W(rows)”即读写多行（实则是扩展 1.1.3 节 SQL 标准提出的不可重复读的定义），即对多行的不可重复读。只是 ANSI SQL 标准着眼于在单行上定义不可重复读，本节扩展的定义着眼于在多行上重复单行上定义的不可重复读（而编码实现的实践中，数据库引擎是对多行数据使用相同的方式进行处理的）。

表 1-10 幻象的写操作改为更新则等同于不可重复读

	不可重复读		幻象	
	P2 ( “Non-repeatable read” )		P3 ( “Phantom” )	
时间	T1 主事务	T2	T1 主事务	T2
t0	R(row)		R(rows)-WHERE<condition>	
t1		W(row)-Update/Delete		W(rows)-Insert/Update =><condition>
t2		Commit	R(rows)-WHERE<condition>	
t3	R(row)			

另外，幻象异常现象中的操作都带有“<condition>”，而不可重复读现象中则没有“<condition>”，这可以把不可重复读变形为类似“R(rows)-WHERE<true>”，即“<condition>”相当于“<true>”。但是，幻象的定义，强调的是在特定的行（元组）被操作后，又有新的行被其他事务操作而产生，所以，我们可以重新校正表 1-9 得到新的表 1-11，发生变化的内容使用深色背景表示，幻象现象中事务 T2 的操作是插入或更新。

从表 1-11 可以看出，不可重复读现象中事务 T2 着眼于对现有数据进行操作；而幻象现象中事务 T2 着眼于对新增（或不在锁定范围内已经存在的数据上做更新后而得的数据满足了谓词条件）数据。这其实正是 Q4 的答案。

Q6：表 1-11 中，不可重复读现象中的事务 T2 在 t2 时刻执行“Commit”或不执行“Commit”会有什么差别吗？或者对于幻象现象，事务 T2 在 t2 时刻没有执行“Commit”，这一点与不可重复读有差别吗？

答：

对于这个问题，我们暂时不做回答，请读者先行思考。在我们讨论了并发控制相关的多种技术后，我们在 2.2.6 节再深入探讨这个问题。





表 1-11 第二次修正后的 SQL 标准定义的三种读数据异常现象

	脏读		不可重复读		幻象	
	P1 ( “Dirty read” )		P2 ( “Non-repeatable read” )		P3 ( “Phantom” )	
时间	T1 主事务	T2	T1 主事务	T2	T1 主事务	T2
t0		W(row)- Update/Insert	R(rows)- WHERE<true>		R(rows)-WHERE <condition>	
t1	R(row)			W(rows)-Update/ Delete-WHERE<true>		W(rows)- Insert/Update =><condition>
t2		Abort		Commit	R(rows)-WHERE <condition>	
t3			R(rows)- WHERE<true>			

## 1.2 事务处理技术的原理

上一节我们讨论了在没有事务模型的情况下，同一份数据因并发操作而出现的各种异常现象（注意，写偏序异常发生在不是同一份数据的并发操作下）。本节我们来讨论解决上述各种异常现象的事务模型及其相关技术，从整体上理解事务的概念以及相关的技术之间的联系。

### 1.2.1 什么是事务

首先，我们来讨论：

**什么是事务？**

面对这个问题，我们尝试掩卷自问，再次看看自己是不是能够很清晰地回答这个基础问题。是不是我们在想：事务，就是 ACID 吧。

谈到事务，言必称 ACID，似乎这成为一个惯性，几乎 ACID 就是事务的代名词。

《Database System Concepts》书说：构成单一逻辑工作单元的操作集合称作事务。

《An Introduction to Database Systems》书说：事务是一个逻辑工作单元。

上面的两种表述，认为事务是数据库关系系统中的一系列操作的一个逻辑单位。

ANSI SQL 标准不仅认为事务是一系列操作的一个逻辑单位，而且明确提出事务“要么成功要么失败”的两种状态：An SQL-transaction (transaction) is a sequence of executions of SQL-statements that is atomic with respect to recovery. That is to say: either the execution result is completely successful, or it has no effect on any SQL-schemas or SQL-data.

Jim Gray 在上述的基础上，还提出事务的 ACID 四个特性：A transaction is a group of SQL commands whose results will be made visible to the rest of the system as a unit when the transaction commits --- or not at all, if the transaction aborts. Transactions are expected to be



atomic, consistent, isolated, and durable.

如果把事务简单认为就是 ACID 四个特性, 有失偏颇, 但是, ACID 确实是数据库管理系统的四根擎天博玉柱, 这四根柱子撑起了数据库管理系统“对数据安全操作的殿堂”<sup>①</sup>。

在数据库引擎内部, 讨论事务, 实则是讨论 ACID。什么是 ACID 四个特性? 作为数据库内核的开发者, 怎么实现这四个特性? 作为数据库的使用者, 您是否知道这四个特性为什么能保证数据的一致性?

我们来讨论 ACID<sup>②</sup>。

维基百科对 ACID 的定义如下 (注意我们使用粗体和背景色标识出了关键词):

- ❑ **Atomicity** : Atomicity requires that each transaction be "all or nothing": if one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.
- ❑ **Consistency** : The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.
- ❑ **Isolation** : The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on the concurrency control method (i.e., if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.
- ❑ **Durability** : The durability property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

① 关系代数撑起了关系型数据库管理系统数据存储模型、数据计算的DQL模型; 事务模型则保障了整个数据库系统安全、有序运行, 使得数据的一致性、持久性得到保障。

② ACID定义源自: <https://en.wikipedia.org/wiki/ACID>





对于原子性 (Atomicity), 上面的定义强调了两种结果, 或是 “all” 或是 “nothing”, 即事务要么成功要么失败; 两种结果对应的是事务的两种状态, 或是提交 “Committed” 或是放弃 “Aborted”。另外, 还强调了作为事务的一部分 (一个逻辑工作单元中的一个操作) 如果失败则整个事务应该是 “nothing” 的, 换句话说, 事务应 “一损俱损” 但不是 “一荣俱荣”。

对于一致性 (Consistency), 总是一个令人困惑的话题。什么样的才算是一致? 一个人, 言行一致? 一组数据, 数据与数据之间保持一致? 还是如上面的定义所强调的——“from one valid state to another”? 事务的操作使得 “特定的数据” 其状态发生变迁, 但前后的结果对应的状态一直是 “valid” 的。那么, 什么才是 “valid” 的? 把 salary 列的值由 1 改为 2 就是 “valid” 的吗? 答案是: 数据在事务的操作下, 一直符合 “all defined rules” 而这样的规则通常是 “constraints, cascades, triggers, and any combination thereof”, 即这些规则是数据的 “逻辑语义”, 比如, 1.1.5 节提到的写偏序异常, 违反的就是特定数据间的 “constraints”。约束, 属于用户的语义所限定的数据的一致性。对于一个数据库系统, 一致性的另外一层含义, 属于系统级, 是指要想使得数据在数据库系统中保持一致, 还要求数据库系统符合两个特性, 一是 “可串行性 (serializability)” 二是 “可恢复性 (recoverability)”。可串行性很重要, 在隔离性下定义, 本节下一段描述。可恢复性是指已经提交的事务未曾读过被回滚的事务写过的数据 (注意不是指数据库系统宕机重启后所做的恢复操作, 而是不会发生 “脏读”), 这个特性也很重要, 当事务回滚, 被回滚的事务就不应当对数据的一致性造成影响, 数据库的一致性状态是可恢复的。所以, 可串行性保证了数据不被并发操作改坏, 可恢复性则保证了事务被回滚后数据回到之前的 “valid” 状态<sup>①</sup>。

对于隔离性 (Isolation), 上面的定义强调了一个 “concurrent execution”, 这是指存在多个事务 (多个会话中的不同的但同一时间段内运行的事务) 同时运行, 但是他们运行的顺序就好像是 “serially”, 这意味着 “并发” 看起来像是 “串行”, 但不是说 “并发” 的这些个动作确实是 “串行” 的, 而是说 “并发” 的这些个动作对数据的操作之后的数据的最终状态应该是 “valid” 的, “valid” 的数据看起来像是 “串行” 的那些个动作造成的。另外, 可串行化隔离性定义的是 “serializability”, 不要把隔离性与 “隔离级别” 以及 “隔离级别” 中的可串行化 “SERIALIZABLE” 相混淆, “隔离级别” 是为了提高并发度、从而弱化了数据在并发读写下的 “一致性” (如写偏序)、且允许出现如 1.1 节所述的若干种异常而提出的、几种和数据一致状态有关的、几个避免不同数据异常现象的数据一致性层级, 这个层次的最高层 “SERIALIZABLE” 才能做到 “隔离性”。

对于持久性 (Durability), 上面的定义强调的是对于 “committed” 状态的数据, 要能够永久保存, 这除了物理存储外, 还需要防止处于 “committed” 状态的数据因数据库引擎没有来得及把数据保存到物理存储上 (如掉电) 而丢失了已经被 “commit” 的数据。所以, 从持久性的定义看, 在原子性中提到的 “aborted” 状态的数据, 持久性是不关注的。

① 数据库系统实现时, 需要保证: 对于并发的事务  $T_i$  和  $T_j$ , 如果  $T_j$  读取了由  $T_i$  所写的的数据项, 则  $T_i$  应先于  $T_j$  提交。如果不这样, 则当  $T_j$  先提交, 但  $T_i$  被回滚, 则事务  $T_j$  不再可能被恢复。



上面我们讨论了 ACID 四特性，正是这四个特性的提出，才使得事务的概念丰满起来、形象起来；同时也激发了很多针对这四个特性的技术，这是 1.2.3 节要讨论的内容。但是在讨论并发控制技术之前，还需要明确事务的一些重要属性，这些属性影响着事务的并发控制技术（参见 1.2.5、2.2 节），所以需要特别重视，下一节将重点讨论这些重要属性。

## 1.2.2 事务的属性

在讨论事务的属性前，我们先明确几个与事务的操作相关的基本概念：

- ❑ 提交 (Commit)：事务的动作，表示事务成功执行。
- ❑ 中止 (Abort)：事务的动作，表示撤销事务的执行。仅仅是一个事务的动作，但此动作会引发回滚的执行。
- ❑ 回滚 (Rollback)：事务的动作，表示撤销事务的执行，但同时又有进一步的动作，分为两种情况：
  - 第一种情况意在恢复被修改的数据：恢复旧的数据值使得数据的一致性得到保证。中止这个事务动作实际上可能会引发恢复操作，但不一定所有被中止的事务都需要执行恢复操作，如只读事务被中止就不必要做恢复。所以更新事务写操作发生后的中止操作需要“回滚”。
  - 第二种情况意在表示被中止的事务在中止后，此事务被事务调度器自动重启（事务被重新赋予新的时间戳值然后开始重新执行），这种情况用于基于时间戳的排序并发控制技术中。
  - 所以，本书用到回滚一词的时候，是区分语境的，请多注意。多数情况下其含义指第一种。

如果说 ACID 是事务的特性<sup>①</sup>，那么，可串行化 (Serializability)、可恢复性 (Recoverability)、严格性 (Strictness) 则可以称为事务的属性<sup>②</sup>。

如下我们分别讲述事务的这些属性<sup>③</sup>。

### 1. 可串行化 (Serializability)

维基百科上对可串行化的英文释义如下：

Serializability of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.

可串行化是一个调度，即多个事务之间的执行方式；而多个事务之间的执行有个先后

① 特性：某事物所特有的性质

② 属性：就是对于一个对象的抽象刻画

③ 本节涉及的英文定义源自：<https://en.wikipedia.org/wiki>





顺序, 如果事务之间没有共同的操作对象(读或写操作), 则事务之间的执行顺序前后置换是没有关系的; 但是如果事物间存在共同的操作对象, 则事务间先后执行的顺序则需要区分; 对于存在共同操作对象的多个并发执行的事务, 如果其执行结果“等价”于某个“串行化调度”, 则这个调度才是“可串行化的调度”。满足“可串行化的调度”则具有了可串行化 (Serializability) 属性。因此, 可串行化 (Serializability) 属性保证的是多个事务并发时的执行顺序要对数据的一致性没有影响。

如表 1-12 所示, 三个并发的事务, 之间满足串行化调度 (serial schedule), 即事务之间没有操作相同的数据 (分别操作了不同的数据 X、Y、Z), 而可串行化调度的执行结果, 调度结果等价于串行化调度, 但需要引入几个概念来保证等价效果, 这几个概念就是: 冲突行为 (Conflicting actions)、冲突等价 (Conflict equivalence)、冲突可串行化 (Conflict-serializable)。

表 1-12 串行化调度示例

T1	T2	T1
R(X)		
	R(Y)	
		R(Z)
W(X)		
	W(Y)	
		W(Z)
Commit	Commit	Commit

下面我们来讨论冲突行为、冲突等价、冲突可串行化。

□ 冲突行为 (Conflicting actions): 又称为冲突动作, 当有两个动作满足如下三个条件, 则这两个动作是冲突的:

- 此两个动作属于不同的事务;
- 至少一个动作是写操作;
- 动作在操作同一个对象 (读同一个对象或写同一个对象)。

□ 冲突等价 (Conflict equivalence): 对于不同的事务调度方式 S1 和 S2, 如果满足如下两个条件, 则事务调度方式 S1 和 S2 是等价的:

- S1 和 S2 调度方式包括同样的事务集合 (每一个事务中的操作的顺序是固定的, 不能在不同的调度方式下发生变化);
- S1 和 S2 调度方式包括同样的冲突操作集合。

□ 冲突可串行化 (Conflictserializable): 当某个调度是一个“冲突等价”于一个或多个“串行调度”, 则这个调度是“冲突可串行化”的。这相当于把并发事务等价于了某“多个事务的串行执行”, 用显而易见的串行的结果必然满足一致性 (数据的一致性) 来表示并发事务的调度带来的结果也满足一致性 (所以事务可以因此而并发地被执行以提高执行效率)。如表 1-13 所示, 并发的调度方式中包括事务 T1 和 T2 在同时执行, 这个调度方式“冲突等价”于串行执行的“<T1,T2>”而等价于“<T2,T1>”, 尽管只有一个“冲突等价”的串行执行, 这个调度也是“冲突可串行化”的。

表 1-13 冲突可串行化调度示例

T1	T2
R(X)	
	R(X)
W(Y)	
Commit	
	W(X)
	Commit

所以，可串行化概念的作用在于保证并发的事务调度方式既能满足数据一致性需求，又能提高并发事务的执行效率。

对于不同的并发控制方法，基于锁的方法是通过阻塞其他产生冲突的事务来保证可串行化的；基于时间戳的方法通过回滚产生冲突的事务来保证可串行化；基于有效性检查的方法本质上是基于时间戳的并发控制方法，所以它也是通过回滚产生冲突的事务来保证可串行化的。

另外，可串行化的概念之外还有“视图等价”（View equivalence）和“视图可串行化”（View-serializable）这两个概念，因为“视图可串行化”（View-serializable）是一个“NP-complete<sup>⊖</sup>”问题而没有实用价值，本书不再描述，可参见相关书籍。

## 2. 可恢复性 (Recoverability)

维基百科上对可恢复性的英文释义如下：

Recoverability means that committed transactions have not read data written by aborted transactions (whose effects do not exist in the resulting database states)。

可恢复性是并发的事务后期、表明提交阶段事务间相互影响的属性。即：已经提交的事务没有读过被中止的事务的写数据。否则脏读异常发生，导致数据不一致。所以可恢复性属性保证的是多个事务并发调度后期的提交顺序对数据的一致性没有影响。

如表 1-14 所示，case 1 的事务 T2 读数据 X 发生在事务 T1 写数据 X 之后，两个事务都提交，没有违反“可恢复性”的定义；case 2 的事务 T2 读数据 X 发生在事务 T1 写数据 X 之后，事务 T1 和 T2 都被中止，没有违反“可恢复性”的定义；所以 case 1 和 case 2 这两种调度方式都具有“可恢复性”。而 case 3 发生了脏读，所以不满足“可恢复性”。

表 1-14 可恢复性示例

可恢复				不可恢复	
case 1		case 2		case 3	
T1	T2	T1	T2	T1	T2
R(X)		R(X)		R(X)	
W(X)		W(X)		W(X)	
	R(X)		R(X)		R(X)
	W(X)		W(X)		W(X)
Commit		Abort			Commit
	Commit		Abort	Abort	

前面我们说：可恢复性属性保证的是多个事务并发调度后期的提交顺序对数据的一致性没有影响。那么，如果可恢复性不被满足，即脏读发生，如表 1-14 的 case 3，数据库系统的事务管理器应该怎么处理呢？答案是：暂停“提交”或读写操作，并根据之前的相冲突的事务的状态决定本事务的状态。即如果是封锁机制，则事务 T1 的写操作抑制了事务 T2 的读操作；如果是 MVCC 机制，则事务 T1 的写操作抑制了事务 T2 的写操作，致使事

⊖ Non-Deterministic Polynomial Complete Problems



务 T2 回滚；如果是乐观机制，则事务 T2 的读写操作可继续执行，但是否能提交取决于事务 T1 是提交还是回滚。

由此又引发一个新的问题，称为“避免级联中止”<sup>①</sup>（Avoids cascading aborts / rollbacks (ACA)），即事务管理器在做事务调度的时候，就要避免 case 3 的情况发生，以避免因为事务 T1 被中止而导致事务 T2 被迫被中止掉。如果事务 T2 被迫被中止掉，则称为事务 T2 因事务 T1 的中止而被牵连即发生级联回滚。

我们再来看表 1-14 中的 case 2，事务 T1 被中止，但事务 T1 写数据 X 在事务 T2 读数据之前，所以事务 T1 发生中止导致事务 T2 不得不中止，这样才能保证数据的一致性。这说明尽管某个调度如 case 2 符合可恢复性，但是还是可能存在级联中止的可能。而级联中止会为数据库系统的事务管理器带来额外的操作而影响效率，事务管理器应当在进行事务调度的时候，就采取措施避免发生级联中止。所以，“避免级联中止”成为了比“可恢复性”更为严格的一个事务属性。

### 3. 严格性 (Strictness)

维基百科上对严格性的英文释义如下：

A schedule is strict - has the strictness property - if for any two transactions T1, T2, if a write operation of T1 precedes a conflicting operation of T2 (either read or write), then the commit or abort event of T1 also precedes that conflicting operation of T2.

所以，严格性概念的作用在于保证：有冲突动作（前述的“冲突行为（Conflicting actions）”）的并发的事务中，先发生写操作的事务提交或中止的操作优先于其他事务。这个属性确保了并发的事务间的提交 / 中止这个动作的顺序。这个属性也是在编码实现事务管理器时的一个重要的细节点。

如表 1-14 所示，事务管理器保证只能发生 case 1 和 case 2 的情况，而不能发生 case 3 的情况。但编码实现时应该在事务提交或中止动作发出的时候，判断这个动作是否是由事务 T1 发出的，判断事务 T1 是不是先做过写操作，如果是，才允许事务 T1 提交或主动中止或被动被中止（当然，事务 T1 之前已经没有“优先于”T1 的其他未完成的事务存在）。对于事务 T2，则需要继续判断事务 T1 是不是已经提交，如果事务 T1 正常提交，则事务 T1 才被允许执行提交或中止操作。

### 4. 各个属性之间的关系

前面讲述了事务的多个属性，这些属性从属于事务的调度方式，属性之间从概念上存在一个语义包含的关系，如图 1-3 所示。

说明：

□ 从串行化的角度看，如图中纵向的方框，串行化的严格程度从松到严的次序为：all

① 特别注意：一些书籍不区分 Abort 和 Rollback 的差别，把他们当作一个含义。所以常把“Avoids cascading aborts / rollbacks (ACA)”称为“避免级联回滚”。但是，本书称其为“避免级联中止”，意在区分 Abort 和 Rollback 的差别。

schedules  $\rightarrow$  view serializable  $\rightarrow$  conflict serializable  $\rightarrow$  serial。

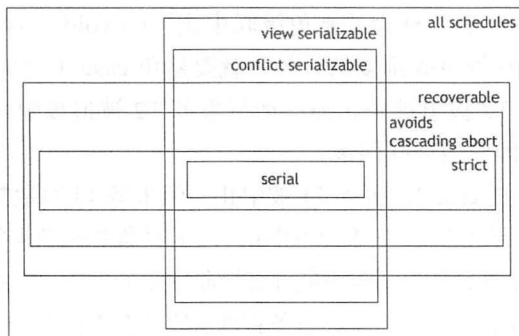


图 1-3 各个属性之间的关系图<sup>①</sup>

- 从可恢复性的角度看，如图中横向的方框，严格程度从松到严的次序为：recoverable  $\rightarrow$  avoids cascading aborts  $\rightarrow$  strictness  $\rightarrow$  serial。
- 从图中可以看出，所有的事务调度方式中，最核心的是“serial”，“serial”能够保证数据的一致性，但是其执行效率低。所以在确保数据一致性的前提下，不同的调度方式致力于提高并发执行的效率，使得数据库的事务管理器能够高效地运行。
- 在 1.2.5 节介绍的各种并发技术，就是致力于保证在并发的调度正确性的前提下，提高事务管理器的事务并发调度的效率。多种并发控制技术含义就是针对并发事务的不同的调度方式。

### 1.2.3 ACID 的实现技术

各种数据库理论的教科书，都会详细讨论 ACID 的实现技术，如并发控制、日志管理、备份恢复、锁的管理、MVCC 等内容。

这些内容和 ACID 的实现紧密相关，所以每一个教科书都不厌其烦地对这些技术加以论述，而且都是以大量的篇幅来探讨这些技术。初入门者，很容易坠入其中而不能“自拔”，即不能从整体上对 ACID 的实现技术进行把握。比如说，我们提出一个问题如下：

请简洁地总结一下哪些技术是分别实现 ACID 中的 A 的？哪些是实现 C、I、D 的？

您若能掩卷细思，不妨把答案写在纸上，一并对比如下内容。

实现 ACID 的核心技术参见图 1-4。

从图 1-4 可以看出，实现 ACID 的核心技术，是并发控制和日志相关技术，在下面的 1.4 节和 1.5 节中我们将分别简要地探讨实现 ACID 的主要技术，探讨的过程主要侧重于与事务的关联点。

<sup>①</sup> 源自：[https://en.wikipedia.org/wiki/Schedule\\_\(computer\\_science\)#Commitment-ordered](https://en.wikipedia.org/wiki/Schedule_(computer_science)#Commitment-ordered)。



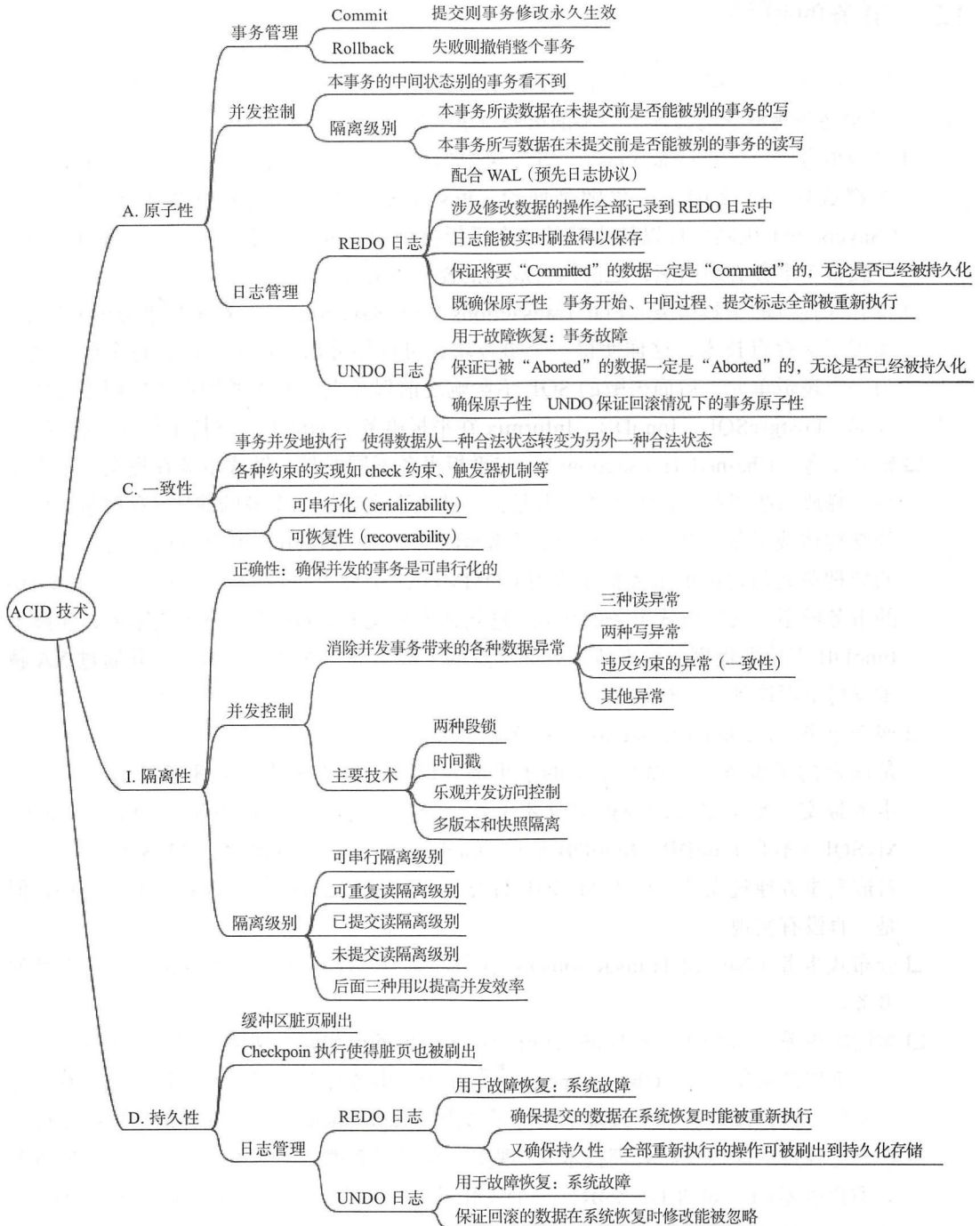


图 1-4 实现 ACID 的核心技术图

## 1.3 事务的模型

事务的实现，在不同的数据库系统中是不同的，这是因为事务有着不同的模型，在 Jim Gray 的《事务处理概念与技术》一书的第四章事务模型中，事务被分为：

- ❑ 平板事务 (Flat Transactions)：事务块中的所有 SQL 语句，构成一个逻辑单元，要么都成功，要么因之一失败都回滚。PostgreSQL 的事务管理如果不考虑保存点 (Savepoint) 机制，可以认为就是一个平板类型的事务，事务块内的一个 SQL 失败，导致整个事务必须回滚，之前执行成功的操作也必须回滚掉。
- ❑ 带有保存点的平板事务 (Flat Transactions With Savepoints)：在平板事务的基础上，实现了保存点技术，这样使得一个事务块，可以划分出不同的层次，每个层次之间为一个逻辑单元，后面失败的 SQL 不影响之前保存点前发生的操作，即回滚发生在局部。PostgreSQL、InnoDB、Informix 在平板事务的基础上，支持了保存点技术。
- ❑ 链式事务 (Chained Transactions)：与平板事务不同的是，链式事务在提交一个事务后，释放一些资源如锁等资源，但是，一些上下文环境如事务的载体（存放事务信息的结构体或类等对象）不被释放，会留给下一个事务使用。用户感觉自己的逻辑上的处理单元与之前的事务似乎没有 COMMIT 之类命令执行的明显分割。如 InnoDB 的事务模型，就是链式事务的代表（这句话不是说 InnoDB 不支持平板事务，实际上 InnoDB 支持平板事务、支持带有保存点的平板事务、支持链式事务，并通过 XA 技术支持下面谈到的分布式事务）。
- ❑ 嵌套事务 (Nested Transactions)：嵌套事务如同一棵树，树有子叉，每个子叉可以是嵌套的子事务也可以是平板的子事务。但叶子节点的事务是平板事务。根节点事务提交，整个事务的数据修改才生效，否则只是事务内局部有效。PostgreSQL、MySQL（不是 InnoDB，InnoDB 是被 Oracle 收购之后才逐渐并入 MySQL 的）没有对嵌套事务通过支持。原本 MySQL 打算在 5.0 版本之后提供对嵌套事务的支持，但是一直没有实现。
- ❑ 分布式事务 (Nested Transactions)：在分布式环境下的平板事务或以上其他类型的事务。
- ❑ 多层次事务 (Multi-Level Transactions)：多层事务也如同一棵树，树根是事务的总节点，下层是对象操作 (Object Operation) 作为子事务存在，对象操作还可以带有子对象操作节点，或带有一个或多个的叶子节点 (Page operation)。这样的事务模型可以有自己独特的并发控制处理技术<sup>⊖</sup>，现实中有工程实现的不多见，而且与之前的事务分类角度不同。如图 1-5 所示，一个简化的两层事务，从叶子节点起为 level 0，逐层向上编号。

⊖ 更多详细信息参见论文《Multi-Node Multi-Level Transactions》。

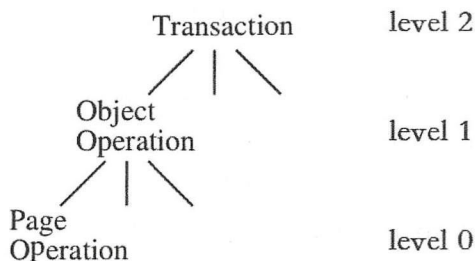


图 1-5 多层事务的一个简化版本（两层事务）

这些不同的模型，在不同数据库中的实现方式是不相同的。尽管每个数据库都遵循相同的原理，但实现各有不同，这点能体现出理论和工程的差异性（有的时候这些差异是非常大的），掌握这些差异性，对于数据库内核开发的工程实践人员尤为重要。尤其对于数据库内核的架构设计者，体悟、领会、掌握不同数据库引擎在相同模块的处理方式上的差异，对于指导自身进行设计更有实战意义。

本书第三、第四篇会以实例来展示事务管理和并发控制技术范围内的差异，阅读时，建议多和理论部分做纵向对比，也在不同数据库引擎之间多做横向对比。

另外，本书行文时，也将努力加强横向和纵向的对比。例如，在第十章讲述 InnoDB 事务管理技术的时候，本书将对比 InnoDB 的 Mini-Transaction 和 Informix 内部的事务管理技术，以帮助读者扩展思维，更好且更深地掌握相关内容。

## 1.4 并发控制技术

并发控制技术是实现原子性、一致性和隔离性的重要技术之一，是本书的重点。

本节从整体上概述并发控制技术的思路和所用的方法，对并发控制技术更为详细的介绍，请参看 2.2 节。

并发控制技术的本质，就是要对并发的事务实现正确的（保证数据的一致性、保证事务操作的原子性）、高效的（用“可串行性 / 可恢复性 / 严格性”实现可并发，部分情况下牺牲一致性，或用低级别的隔离性容忍不一致以提高并发执行效率）调度。

并发控制技术，就是对并发操作进行“控制”的技术，“控制”的含义就是进行限制，即限制并发以保证数据的一致性。串行化的含义是完全限制并发；可串行化是在能保证一致性的情况下，允许某些并发的操作被执行；以提高数据整体的运行效率。

### 1.4.1 并发控制技术的实现策略

并发控制技术，从实现的思路角度看，有两类<sup>①</sup>，就是我们常说的乐观与悲观并发控

① 另外一种称为 Semi-Optimistic，可参考论文：*A Semi-Optimistic Database Scheduler Based on Commit Ordering*。



制，这两类思路的差别在于，是事后检查还是提前预防。

□ 乐观 (Optimistic concurrency control, OCC)<sup>①</sup>：从一开始，每一项操作都允许进行，但在事务提交的时刻，进行隔离性和完整性约束的检查，如果有违反则事务被中止。显然，如果并发冲突少的场景，乐观并发控制方法是适合的。2.2.3 节讨论的基于有效性确认的并发控制方法就是乐观的。

□ 悲观 (Pessimistic concurrency control, PCC)：从一开始，即检查每一项操作是否会违反隔离性和完整性约束，如果可能违反，则阻塞这样的操作。如两阶段封锁，用读锁来阻塞另外一个事务的写锁。就是因为另外一个事务的写操作可能会造成如 1.1.3 节提到的读异常，因此两阶段封锁技术属于悲观的方法，提前对异常现象进行了预防。基于时间戳排序的并发控制技术也是悲观的。

### 1.4.2 并发控制技术的实现技术

并发控制技术，从实现的技术角度看，有很多种，主要的一些技术方向，包括：基于时间的 (Time-stamp ordering)、基于提交顺序的 (Commitment ordering)、基于串行化图测试验证的 (Serialization graph testing)、基于锁的 (Locking)，如下：

□ 时间戳 (Timestamp ordering, TO)：基于时间戳对事务提交顺序排序的并发控制技术。

- 首先，时间戳排序技术中有两类主体，一是事务，二是数据项。时间戳就要“盖（赋值）”在这两类主体上。
- 其次，为每个事务分配一个时间值（通常是在事务开始的时候分配，但有的系统是在事务提交的时候才分配时间值）作为此事务发生的标识，这个时间值称为一个“时间戳”，“时间戳”就如同为事务盖了一个章。时间值取值有两种方式，一是系统时钟，二是逻辑计数器。
- 第三，数据项上有两个时间戳，一个是“读时间戳”，记录读取该数据项的最大事务的时间戳，另一个是“写时间戳”，记录写入该数据项当前值的事务对应的时间戳，即最新的修改该数据项的事务的时间戳标识。
- 第四，因存在并发所以通过检查  $T_i$  事务的时间戳和  $T_j$  事务的数据项上的时间戳以确定并发事务  $T_i$  和  $T_j$  之间的先后关系（如果  $T_i < T_j$ ，则事务调度器必须保证所产生的并发调度等价于事务  $T_i$  先于事务  $T_j$  的某个串行调度）。
- 第五，确保任何有冲突的 READ 或 WRITE 操作按时间戳顺序执行（执行的具体方式详见 2.2.4 节）。
- 所以，时间戳并发控制技术用来确保在访问冲突的情况下，多个事务按照时间戳的顺序来访问数据项。如果  $TS(T_i) < TS(T_j)$ ，那么数据库事物管理器必须保证所产生的调度等价于事务  $T_i$  出现在事务  $T_j$  之前的某个串行调度。

① 乐观并发控制方法的思路由孔祥重教授提出：Kung, H.T. (1981). On Optimistic Methods for Concurrency Control.



□ Commitment ordering (或 Commit ordering, CO): 提交排序是对提交操作的顺序进行排序, 这种方式是“冲突可串行化”(参见 1.2.2 节)的一个特例:

- 定义为: Let  $T_1$  be two *committed* transactions in a schedule, such that  $T_2$  is in a conflict with  $T_1$  ( $T_1$  precedes  $T_2$ ). The schedule has the Commitment ordering (CO) property, if for every two such transactions  $T_1$  commits before  $T_2$  commits.
- CO 区别于封锁并发控制技术之处在于在事务结束前, 读或写等操作不互相阻塞, 只是在提交阶段判断是否存在 1.2.2 节讲述的“冲突行为”, 以决定并发地事务的提交顺序。换句话说, CO 只会在提交阶段发生阻塞。因此 CO 可被认为是“乐观 (Optimistic concurrency control, OCC)”的方法。
- CO 的缺点在于这个并发控制方法不具有“可恢复性”(参见 1.2.2 节)。
- CO 被主要用于分布式事务处理, 存在多种变种分布式 CO 的特殊之处在于, 决定是否可以提交的事件是由“a local commitment mechanism”和“an atomic commitment protocol”(下面讨论的 2PC 是一种“an atomic commitment protocol”)来确定的。这表明 CO 被用于协调本地事务 (a local commitment mechanism) 和分布式事务 (an atomic commitment protocol), 即用于在分布式环境下实现分布式事务的, 所以才能有 2PC 是 CO 的一种“an atomic commitment protocol”的说法。

□ 串行化图形检测 (Serialization graph checking): 也称为优先图 / 冲突图 / 串行化图 (Precedence graph、conflict graph、serializability graph) 检测, 主要含义如下:

- 在一个调度  $S$  中, 每一个已经提交的事务都是一个节点。
- 如果事务  $T_i$  优先且冲突于事务  $T_j$ , 则存在一条边从  $T_i$  节点指向  $T_j$  节点。
- 这样, 调度  $S$  中的所有的事务代表的节点和事务间的优先关系构成的边, 构成一个有向图。
- 串行化图形检测, 就是检查上述的图中是否存在环, 如果存在环, 则违反了 1.2.2 节提到的“冲突可串行化 (Conflictserializabl)”即存在“冲突行为 (Conflictaction)”。因此存在环的调度不应当成立。
- 有向边的建立, 存在三种情况, 都是对于同一个数据项进行操作: (1)  $T_i-R(X) \rightarrow T_j-W(X)$ ; (2)  $T_i-W(X) \rightarrow T_j-R(X)$ ; (3)  $T_i-W(X) \rightarrow T_j-W(X)$ ; 这三种情况都存在一条边从  $T_i$  节点指向  $T_j$  节点。这也表明“ $T_i-R(X) \rightarrow T_j-R(X)$ ”读读操作不存在有向边。

□ 两阶段封锁 (Two-phase locking, 2PL):

- 首先, 两阶段封锁强调的是“加锁 (增长阶段, growing phase) 和解锁 (缩减阶段, shrinking phase) 这两项操作, 且每项操作各自为一个阶段”, 这就是说不管同一个事务内需要在多少个数据项上加锁, 所有的加锁操作都只能在同一个阶段完成, 在这个阶段内, 不允许对已经加锁的数据项进行解锁操作, 即加锁和解锁操作不能交叉执行 (同一个事务内)。这一条是说在同一个事务内部的事情。



○ 其次，为了提高并发度，才对锁进行分类，分出共享锁（读锁）和排它锁（写锁），因这两种类型的锁，又产生加两种锁共四种事务因并发受影响的情况：

■ 一是先对数据项加共享锁，此读锁不阻塞其他事务也读取本数据项，就是说读－读并发是允许的，即第一种情况（两个事务并发读同一个数据项）；但是此读锁阻塞其他事务写本数据项，这就是说读－写并发是不允许的，即第二种情况；这里所说的读－读和读－写的第一个读在前，是因；第二个读或写是可能在其他事务发生的操作，是果，前者（第二个读操作）能够发生后者（写操作）不能够发生。

■ 二是对数据项施加了排它锁，这使得其他事务在这个数据项上的读操作（第三种情况）或写操作（第四种情况）都被禁止。

■ 这一条是说在多个事务之间的事情。

■ 多个事务之间比较锁是否兼容，用到了锁的兼容性列表，就是上面的四种情况。

只是当锁的类型被扩展后，增加了意向锁等类型，才使得锁的兼容性列表变大，不再是四种情况，而是更多种。

○ 第三，共享锁是允许向排它锁升级的，排它锁是允许向共享锁降级的，升级（upgrade）和降级（downgrade）操作，称为锁转换（lock conversion）。升级只能发生在增长阶段，降级只能发生在缩减阶段。升降级发生在同一个事务内部，但目的也是为了提高多个事务之间的并发度。同一个事物内部比较锁是否兼容，用到的是锁的升级列表，这与锁的兼容性列表是不同的。

○ 所以，两阶段的含义是指在同一个事务内，对所涉及的所有数据项进行先加锁，然后才对所有的数据项解锁。但两阶段封锁第一阶段加共享锁后影响了其他事务的写操作、加排它锁后影响了其他事务的读操作（读受影响更不用提写），所以较大地影响了其他事务的运行（如果不操作相同数据项则互不影响）。只有第二阶段释放了所有的数据项上的锁之后，才能运行其他要操作相同数据项的事务。

○ 按提交操作的时机不同，两阶段锁可以分为 S2PL 和 SS2PL 两种方式，区别如图 2-5。

并发控制技术，有一些不单独使用，而是配合其他并发控制技术一起使用，用以改善主并发控制技术以提高并发度（所以需要特别注意这些技术的适用范围和缺点，其缺点正是需要使用其他并发控制技术弥补，详细信息参见 2.2 节），如下：

□ 多版本并发控制技术（Multiversion concurrency control (MCC or MVCC)）：事务管理器为写操作生成一个数据项的新版本；当有读操作发生时，按照读操作所在事务开始阶段获得的活动的快照、找出应该读取的该数据项的某个版本。这样，读－写操作或写－读操作不会互相阻塞，只留有写－写操作互相阻塞，进一步提高了并发度。为了进一步提高并发的效率，多种并发控制技术协同使用效果更好。如封锁协议+MVCC，前者只有读－读不被阻塞，后者只有写－写阻塞。这两个协议组合后使得基于封锁协议的读－写、写－读不被阻塞。PostgreSQL 和 MySQL 中的 MVCC





机制就是这样的组合。但是，这里描述的只有早期 MVCC 的技术（这些技术目前在 PostgreSQL、InnoDB 中被广泛使用），其关键点在于解决“写-写”并发冲突，但是这样的 MVCC 技术不能避免“写偏序”数据异常。有一种较新的 MVCC 技术，称为“write-snapshot isolation”的技术，致力于解决“读-写”冲突且能做到“可串行化”，比常规的 MVCC 技术更加提升了并发度。详情可参阅论文《A Critique of Snapshot Isolation》。分布式数据库 CockroachDB 就采用了此篇论文提及的“write-snapshot isolation”技术。

- ❑ 基于索引的并发控制技术（Index concurrency control）：在索引树上对索引页采取封锁手段，以维护索引树的一致性；同时，可以确保避免幻象异常，如 MySQL 的 InnoDB 存储引擎以 B+ 树作为存储的基本结构（即索引组织表），可以在索引树上直接施加“next-key locking”进行范围锁定，以避免在谓词限定的“谓词空间”内新数据被插入或旧数据被删除。

更多详细内容，请参见 2.2 和 2.3 节。

## 1.5 日志技术与恢复子系统

日志技术是有效实现原子性和持久性的重要技术之一。

数据库系统在运行期间，对于一个事务中的每一个 SQL、对于每个 SQL 操作，都不是一下子就能执行完毕的。当操作涉及数据的修改时，意味着数据的一致性状态在发生变迁。为了保证数据状态变迁的过程是安全的，把修改数据状态的操作的对象、数据和过程记载下来，当系统故障发生需要做系统恢复的时候，进行回放，就可以应对事务的原子性（重执行已经提交的事务的全部过程，还原未提交的事务的旧数据）和持久性（刷出数据到存储，包括日志数据提前刷出，脏数据被保存点 Checkpoint 刷出）。

需要记录的数据通常包括：

- ❑ 事务的标识：如事务的 ID，有的数据库系统使用事务发生的时间有的使用一个全局的唯一数字表示事务的标识。
- ❑ 数据项的标识：哪个数据库中的哪个模式下的哪个表的哪条数据项被修改，可以换算成物理存储形式下对应的哪个物理地址的数据项被修改。
- ❑ 旧值：数据项被修改前的值，又被称为前像。
- ❑ 新值：数据项被修改后的值，又被称为后像。

而一系列的操作构成了一个操作序列并带有过程中产生的数据，这就是日志。数据库引擎实现时，日志产生后被保存到日志缓存区，然后被刷出到外存，存放到日志文件中。通常，日志文件可以有多个，多数数据库系统要求日志文件至少有三个以便于在复用日志的时候可以在日志文件间自由切换，这就涉及了日志的管理问题。有的数据库系统，日志文件是不重用的，如 PostgreSQL 的日志文件名称是连续增长的，而使用文件组等概念的数



数据库系统如 Oracle、Informix，它们的日志文件是重用的。不重用的日志文件，能够容忍长事务存在，而重用的日志文件因事务太长导致所有的日志文件用光了的话，会因无日志文件可用导致数据库系统不得不挂起，如 Informix。

日志根据所记载的内容，可以从形式上分为两种，一个是 REDO 日志，一个是 UNDO 日志。REDO 日志记录事务的标识、数据项的标识、新值；UNDO 日志记录事务的标识、数据项的标识、旧值。

在系统运行期间，REDO 日志要求执行 COMMIT 操作的 COMMIT 日志的记录要先于被修改的数据项到达外存（即预写日志技术）。这才能在系统故障发生后，在系统恢复阶段保证事务的原子性（如下面的两种恢复策略）。

在系统运行期间，当事务故障发生时，数据库在执行回滚操作时可以使用 UNDO 日志记录的旧值覆盖数据项的新值。

在系统故障发生后，系统恢复期间，通常要使用 REDO 和 UNDO 日志来做恢复，这时，有两种策略可以选择：

- 策略一：恢复时，使用 REDO 日志只重做已经提交的事务，如 PostgreSQL 如此实现，这样省却了 UNDO 日志（PostgreSQL 没有实现 UNDO 日志）。
- 策略二：恢复时，使用 REDO 日志重做所有事务，包括未提交的事务和回滚的事务，然后通过 UNDO 日志回滚那些未提交的事务（未成功的事务一定不能以新值覆盖旧值，即撤销新值对旧值的影响），如 InnoDB 如此实现。

有关日志的相关内容包括很多，本书不进一步加以探讨，相关内容请参见相应书籍和各个数据库引擎的源码。有关恢复相关的内容，尤其是分布式环境下和事务相结合的恢复技术，期待将来能有机会详述。

## 1.6 本章小结

本章初步讨论了事务模型要解决的问题以及所使用的技术。

作为一个引子，从概念上先抛出事务相关的多种问题，意在扩展读者的眼界，使得读者对于事务涉及的问题不仅仅限于人所常谈的丢失更新、脏读、不可重复读、幻象这些内容，而是从多个角度讨论了各种读异常和写异常以及快照隔离技术中紧密相关的写偏序异常，这样才能更为明确、更为全面地为读者建立起一幅数据和异常全景图画。

针对各种读写异常的问题，概要性地给出解决问题的技术，从而帮助读者对事务管理技术建立一个初步但有深度的印象。更详细更深入的并发控制技术以及他们之间的关系，参见第 2 章。





## 第 2 章

# 深入理解事务管理和并发控制技术

在 1.2 节讲述了事务原理的基本内容，尤其 1.2.2 节是理解数据库实现 ACID 的重要基础，需要熟记。本章将进一步深入探讨事务相关的内容，结合数据库源码级的实现技术，来深入理解事务原理。

## 2.1 在正确性和效率之间平衡

事务的 ACID，要求并发的事务不能破坏数据的一致性，这个非常重要，这是数据一致性（C 特性）的要求。为了满足数据一致性，才提出了并发控制的技术，而理解并发控制技术，首先可以从使用锁的协议入手，讨论影响数据一致性的三种情况（不同事务之间对同一个数据项的读 - 写、写 - 读、写 - 写这三种情况。读 - 读不影响一致性不必考虑）。然后考虑并发的效率问题，采用多种并发控制技术、适度逐渐放宽并发度<sup>⊖</sup>，从而理解各种各样的并发控制技术。

另外，为了提升数据库处理事务的效率，除了采用不同的并发控制技术逐渐释放事务间的并发能力之外，还有一种思路就是让用户自己判断所发起的操作是否影响到了数据的一致性，如果用户认为不影响数据一致性，则可以控制数据库的事务管理器，放松对事务调度的限制（程序员必须慎重选择隔离级别），以提高并发程度。这样的技术就是“隔离级别”。与 ANSI SQL 标准定义的四种隔离级别所不同的，但与“隔离”沾边的，是“快照隔离”，为了澄清二者的差异，本书特意把“快照隔离”置于本节。

下面我们先来理解隔离级别，然后在 2.2 节探讨各种并发控制技术，在 2.3 节对各种并发控制技术进行比较，从而理解为了提高数据库的运行效率，各种并发控制技术是如何逐

---

⊖ 2.2节讲述的多种并发控制技术的从前到后的顺序，有一个并发度从紧到松的过程。

步处理读-写、写-读、写-写这三种并发操作的。

2.1.1 隔离级别

在 1.1.3 节，我们深入探讨了并发带来的三种异常现象，这三种异常现象，是在并发情况下发生的，数据库系统使用不同的隔离级别，分别隔离了并发情况下对数据读写的三种异常，使得异常不存在，这就是隔离的含义，即隔离异常现象杜绝了异常现象。

SQL 标准（American National Standards Institute，ANSI）提出四种隔离级别，分别是在三种读数据异常现象前后插入四个点，形成了四种隔离地带，不同程度地禁止了三种读数据的异常现象的产生。参见表 2-1。

表 2-1 隔离级别与三个读异常现象

三个读异常现象		Dirty read		Non-repeatable read		Phantom	
		P1		P2		P3	
四个隔离级别	Read Uncommitted		Read Committed		Repeatable Read		Serializable
	未提交读		已提交读		可重复读		可串行化
并发控制粒度	粒度最粗，见到每一个物理元组 <sup>⊖</sup> 都返回给上层，认为元组可见		除了没有提交的元组不返回给上层外，其他都返回给上层		使得本事务不受其他事务的写操作影响		粒度最细，根据事务发生的先后顺序严格判断元组的可见性

Jim Gray 先生对事务隔离性（Isolation）的定义为：

Concurrently executing transactions see the stored information as if they were running serially (one after another).

在 1.2.1 节，我们给出过隔离性的描述如下：

Isolation：The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. Providing isolation is the main goal of concurrency control. Depending on the concurrency control method (i.e., if it uses strict - as opposed to relaxed - serializability), the effects of an incomplete transaction might not even be visible to another transaction.

并发执行的事务能够看到存储的信息（即数据），好似事务之间是串行执行（生成的数据）。这样，因为等价于串行执行，所以数据的一致性不会被破坏，但是这个“隔离性”是隔离级别中最高的级别，其他的隔离级别不能保证“一致性”但却能提高事务的运行效率。

SQL 标准对于隔离级别定义如下：

⊖ 元组（tuple），表示数据库的行存储格式的记录（record），由若干个字段组成。其中包括系统创建的隐含字段和用户自定义的业务相关字段。本书涉及记录的概念，一律用元组表示。



1) Serializable (串行化, 或称为可串行化): 以物理上是可串行化的机制保证逻辑上符合串行化调度。一个事务在执行过程中完全看不到其他事务对数据库所做的更新(事务执行的时候逻辑上不允许别的事务并发执行, 只能通过并发控制技术在避免事务冲突的情况下, 物理上允许多个事务是“可串行化”地执行)。

2) Repeatable Read (可重复读): 一个事务在执行过程中可以看到其他事务已经提交的新插入的元组(读已经提交的, 其实是读早于本事务开始且已经提交的), 但是不能看到其他事务对已有元组的更新(即晚于本事务开始的), 并且, 该事务不要求与其他事务是“可串行化”的。

3) Read Committed (已提交读): 一个事务在执行过程中可以看到已经提交的其他事务新插入的元组, 而且能看到已经提交的其他事务对已有元组的更新。

4) Read Uncommitted (未提交读): 一个事务在执行过程中可以看到没有提交的其他事务新插入的元组, 而且能看到没有提交的其他事务对已有元组的更新。

这四种隔离级别, 都不允许“脏写(dirty write, 参见1.1.4节)”发生。所以在编写程序的时候, 一是要在事务的处理时根据操作是查询还是插入或更新做出判断, 二是不管隔离级别是哪一种, 都要在事务的调度表中禁止脏写发生。如果使用的是MVCC机制, 还要考虑元组的可见性。

下面以并发控制技术采用封锁的方式来概要地了解一下并发控制技术和隔离级别之间的关系, 如表2-2所示。

表 2-2 各种隔离级别在基于锁的并发控制技术下的实现情况<sup>①</sup>

隔离级别	实现方式	不能解决的问题
SERIALIZABLE (可串行化)	添加范围锁(比如表锁, 页锁等)或者如InnoDB在记录锁上添加间隙锁或者用读锁阻塞其他操作等, 直到事务T1结束。以此阻止其他事务T2对事务T1使用WHERE子句限定的范围内的数据(如果无WHERE子句可以认为是WHERE TRUE)进行insert、update等操作	幻读、不可重复读、脏读问题都不会发生
REPEATABLE READ (可重复读)	对于读出的记录, 添加共享锁直到事务T1结束。其他事务T2对这个记录的修改会一直等待直到事务T1结束。但允许其他事务读取同样的数据	当执行一个范围查询时, 可能会发生幻读
READ COMMITTED (已提交读)	在事务T1中读取数据时对记录添加共享锁, 但读取结束立即释放。其他事务T2对这个记录的修改会一直等待直到A中的读取过程结束, 而不需要整个事务T1的结束。所以, 在事务T1的不同阶段对同一记录的读取结果可能是不同的	不可重复读
READ UNCOMMITTED (未提交读)	不添加共享锁。所以其他事务T2可以在事务T1对记录的读取过程中修改同一记录, 可能会导致A读取的数据是一个被破坏的或者说不完整不正确的数据。另外, 在事务T1中可以读取到事务T2(未提交)中修改的数据。比如事务T2对R记录修改了, 但未提交。此时, 在事务T1中读取R记录, 读出的是被B修改过的数据	脏读

① 有关隔离级别与基于锁的并发控制技术间, 封锁与解锁的时机更为直观的对比, 参见2.4.1节。





## 2.1.2 快照隔离

2.1.1 节，我们讨论了 SQL 标准规定的四种隔离级别，本节，我们将讨论这四种隔离级别之外的一种常用的隔离技术，称为“快照隔离 (Snapshot Isolation, 简称 SI)”。“快照隔离”没有被 ANSI SQL 标准定义<sup>①</sup>。

请注意，“快照隔离”与“隔离级别”相同的地方在于“隔离”，即多事务可以并发执行，让相同的数据可以被并发事务同时操作，在并发执行的事务看起来，同一份数据是“隔离”不相互影响的；但是，“快照隔离”只是一种隔离技术，而没有“级别”，即不像“隔离级别”中的隔离技术分为四类，因而有了四种级别，所以“隔离级别”成为 ANSI SQL 标准规定隔离技术的代名词。

如下我们分为八个方面，来分别讨论“快照隔离”技术。

### 1. 什么是“快照隔离”？

维基百科对“快照隔离”的英文释义<sup>②</sup>如下：

snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database (in practice it reads the last committed values that existed at the time it started), and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.

我们可以认为：

- 使用了快照隔离技术的事务中的所有读操作，读到的数据一定是一致的。
- 上一条实际上避免了 ANSI SQL 标准定义的“幻象 / 幻读”等各种读异常现象（具体分析参见表 2-3）。各种读异常的详细讨论参见 1.1.3 和 2.1.2 节。
- 这样的事务如果没有写 - 写冲突发生，则会提交成功。这意味着，不会发生读 - 写和写 - 读冲突。所以并发效率会更高。
- 第一和第三条结果得以保障，是因为从事务开始时，处于当时的并发事务的状态（多个事务的状态，称为快照，即 snapshot）被保存，利用这个快照可以判断本事务和其他事务之间启动的先后顺序、事务的读写数据情况等，以确定是否存在写 - 写冲突。
- 这就是“快照隔离”技术的本质，其实，也是“快照隔离”技术要达到的目的。

---

① 这一点常被人诟病，因为在早期快照隔离被认为是不会存在 ANSI 所定义的三种读异常现象（参见 1.1.3 节）的，所以很多人认可快照隔离。但后来发现用多版本配合快照隔离实现的并发控制技术，会带来写偏序的问题（参见 1.1.5 节），所以之后就推出了“可串行化的快照隔离”以避免写偏序的问题。而 ANSI 提出的四种隔离级别，是早期各大数据库厂商实现并发控制时多采用基于封锁的并发控制技术，所以在基于封锁的技术背景下，才在 ANSI SQL 标准中提出了四种隔离级别。

② 源自：[https://en.wikipedia.org/wiki/Snapshot\\_isolation](https://en.wikipedia.org/wiki/Snapshot_isolation)



表 2-3 修正后的 SQL 标准定义的三种读数据异常现象与快照隔离的比较表

时间	脏读 P1 ( "Dirty read" )		不可重复读 P2 ( "Non-repeatable read" )		幻象 P3 ( "Phantom" )	
	T1 主事务	T2	T1 主事务	T2	T1 主事务	T2
t0		W(row)-Update	R(row)		R(rows)-WHERE <condition>	
t1	R(row)			W(row)-Update/ Delete		W(rows)-Insert/ Update => <condition>
t2		Abort		Commit	R(rows)-WHERE <condition>	
t3			R(row)			
快照隔离的分析	并发事务 T1 和 T2 操作同一个对象的不同版本, 事务 T1 不会受到事务 T2 回滚的影响, 所以快照隔离级别中不存在脏读异常		并发事务 T1 和 T2 操作同一个对象的不同版本, 事务 T2 提交, 事务 T1 不能读取到事务 T2 新提交的数据, 而 t2 时刻的读操作使用了 t0 时刻的同一个快照, 所以快照隔离级别中不存在不可重复读异常		并发事务 T1 和 T2 操作同一个对象的不同版本, 事务 T2 的写操作生成了新的满足 WHERE 子句条件的对象、而不是一个已经存在的对象的新的版本, 事务 T1 不能读取到事务 T2 新生成的对象, 所以快照隔离级别中不存在幻象异常	

明白了上述内容后, 我们就需要来看看, “快照隔离” 的技术实现手段, 是什么?

## 2. “快照隔离” 的技术实现手段

“快照隔离” 是 MVCC 技术的一种实现方式 (详情参见 2.2.4 节)。MVCC 技术的本质, 是为每个对象在写操作发生时, 生成一个新的版本; 在读操作发生时, 读出最近的一个版本。这样同一个元组因有多个版本 (传说中的分身术)、且并发的事务有生命周期存在, 每个事务读到的元组 (实则是元组的某个版本) 是不一样的, 相当于并发事务在操作不同的元组对象, 因此看起来 “不存在并发” (除了写-写冲突一定要操作同一个对象, 如同孙大圣被金钹扣住的一定是真身), 不满足 1.2.2 节提到的 “冲突行为” 的定义。

存在有多个版本的元组级的对象, 物理组织通常有两种: 一是磁盘类型的数据库, MVCC 冗余了磁盘页内部的元组; 二是内存类型的数据库, MVCC 冗余了内存中的元组, 两者所不同的, 是元组存放的位置不同, 前者是位于页面内部, 后者是直接在内存在中组成一个链表。前者典型的实现如 PostgreSQL 和 InnoDB, 后者典型的实现如阿里的 OceanBase、微软的 Hekaton。

## 3. “快照隔离” 的实现技术是如何解决各种冲突的?

□ 首先, 如图 2-1 所示, 有 5 个并发执行的事务, 分别是 T1 到 T5。

□ 其次, 事务所能读取的数据, 一定是已经提交了的事务的、最新的数据。

○ 例如, 事务 T3 只能读取事务 T1 生成的数据, 这是因为事务 T1 的结束时间早于事务 T3 的开始时间; 而事务 T3 不能读取事务 T2 生成的数据, 这是因为事务 T2



的结束时间晚于事务 T3 的开始时间 (事务 T2 和 T3 有时间重叠, 但不能相互获取对方生成的数据, 即避免了不可重复读异常); 事务 T3 也不能读取事务 T4 的数据, 因为事务 T4 的开始时间晚于事务 T3 的开始时间。其他事务可以直接获取数据的条件与事务 T3 是一样的。

- “最新的”含义表明, 假设事务 T1 和 T3 都修改了同一个数据项 X, 则事务 T5 所能看到的数据项 X 一定是事务 T3 生成的版本, 因为事务 T3 生成的版本新于事务 T1 生成的版本。
- 再次, 因为一个数据项有多个版本, 以事务的开始时间戳为获取版本的基准, 并发间的事务看到的是不同的版本。如图 2-1, 假设事务 T1 写了数据项 X, 则初始版本 X0 存在又生成一个新的版本 X1; 此时, 事务 T2 因开始时间早于事务 T1 的结束时间, 所以看不到版本 X1 而只能看到版本 X0; 而事务 T3 因开始时间晚于事务 T1 的结束时间, 所以能看到版本 X1 而不能看到版本 X0; 这样, 对于并发的事务 T2 和 T3, 则分别看到的是同一个数据项 X 的不同版本, 所以这两个事务可以分别读或写 X 的不同版本, 这样的情况, 就意味着读 - 写不被阻塞或者写 - 读不被阻塞。
- 第四, 并发事务同时写同一个数据项, 要遵循 “First-Committer-Wins”, 即: 首个提交者获胜原则。这是在说, 并发的、同时写同一个数据项的事务只能有一个成功, 另外一个必须回滚, 这种情况相当于并发不存在。所以解决了写 - 写冲突。
- 第五, 与 “先提交获胜” 原则相似的是 “先写者获胜”, 即 “First-writer-wins”。

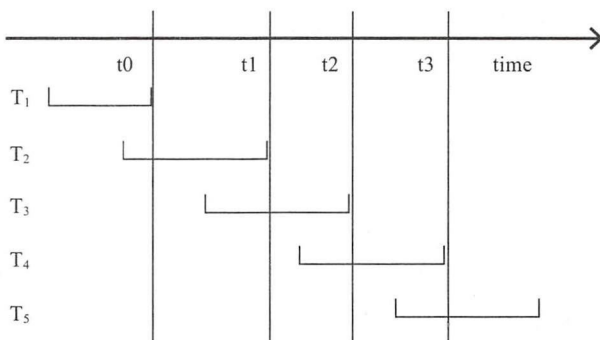


图 2-1 快照隔离中事务之间的关系图

#### 4. “快照隔离” 实现技术的弊端

尽管 MVCC 并发效率很高, 但是 “快照隔离” 不能保证并发事务是可串行化的, 所以, 无法保证数据的一致性。这样的问题, 被称为 “写偏序 (Write skew)”, 参见 1.1.5 节。

#### 5. “快照隔离” 实现技术的弊端的解决方式

使用 “快照隔离” 实现 MVCC 的技术引发了写偏序的问题, 其实是这种技术因多个版本存在使得用户定义在数据项上的语义不能得到保证 (在每个版本上判断语义相当于在不同





的逻辑时空内同时操作不同的分身而不能保证语义施加在同一个物理真身上)而导致的。解决方式是采用“Serializable snapshot isolation”技术,详情参见 2.2.5 节。还有一种技术,叫作“write-snapshot isolation”,曾在第 1 章提及。

## 6. “快照隔离”的其他一些问题

在商业数据库 Oracle (如 10g、11g 这些版本)和开源的数据库 PostgreSQL 中,快照隔离被称为“可串行化(serializable)隔离级别”,这是一种错误的叫法。错误之处在于:

- 一是把“快照隔离”的概念等同了 ANSI SQL 标准中的四种隔离级别中最高的隔离级别。从前述的定义看,显然“快照隔离”和“可串行化(serializable)隔离级别”是明显不同的。
- 二是混淆了二者的真实含义。“快照隔离”不能确保“可串行化(serializable)”这个事务的属性要求;而“可串行化(serializable)隔离级别”却要确保在这个隔离级别下,事务被事务管理器调度时,必须是可串行化的。
- 三是在工程实现中,Oracle 没有在“可串行化(serializable)隔离级别”下真正实现“可串行化”属性。而 PostgreSQL 早期版本也不是一个真的“可串行化(serializable)”的事务,存在写偏序异常;只是在 9.1 版通过“Serializable snapshot isolation”技术实现了真正的可序列化。

## 7. “快照隔离”的实际应用

在 PostgreSQL 和 MySQL 实现中,都使用快照隔离技术实现了 MVCC 协议。其中,PostgreSQL 使用“transaction snapshot”表示快照隔离,MySQL 使用“read view”表示快照隔离,都是用快照来隔离并发事务对相同数据项的操作。

在第二至第四篇中,我们将详细讨论 PostgreSQL 和 MySQL 中 MVCC 和快照隔离技术的实现细节。

## 8. “快照隔离”阅读进阶

- 论文《Database Replication Using Generalized Snapshot Isolation》给出了快照隔离的定义,称之为“Generalized Snapshot Isolation (GSI)”。并基于 GSI 提出了“Prefix-Consistent Snapshot Isolation(PCSI)”用于分布式环境中的复制数据库的事务处理(分布式相关的本书不作讨论,期待有关分布式相关内容将来有机会另述)。
- 论文《Generalized Snapshot Isolation and a Prefix-Consistent Implementation》对 GSI 和 PCSI 做了更多阐述(两篇论文的作者相同)。

## 2.1.3 理解可见性

笔者曾经读到过这么一句话:InnoDB 使用一种称做 ReadView (视图)的对象来判断事务的可见性(也就是 ACID 中的隔离性)。

这句话中提到了“可见性”，但对于可见性的理解，不深刻。下面就笔者的拙见，讨论一下“可见性”。

首先，可见性，是指某个数据项对于不同事务，是否可以被并行操作（即数据项对于某个事务是否可见可操作）。这是从并发的角度而言的，很自然，这就与并发控制算法有关。

其次，为了提高并发度，ANSI SQL 标准定义了四种隔离级别，“SERIALIZABLE”使得数据项不可被其他并发事务可见而被操作（读 - 读除外，但这么说已经和基于锁的并发控制技术关联，因此严格地说，这样的说法也是不严谨的）。其他隔离级别放松了管制，在某些情况下使得数据项被其他事务可见。这表明可见性与隔离级别紧密相关。

在 PostgreSQL 和 MySQL 这样的数据库系统中，因都实现了 MVCC 机制的时候，其中就经常提到“可见性”判断。一个元组对谁可见，意味着这个元组可以被什么样的事务获取（读取、改写或删除），这和数据库引擎使用的并发控制技术、ACID 中隔离性的定义、ANSI SQL 标准定义的隔离级别等紧密相关。

对于基于锁的并发控制技术，如果隔离级别是“SERIALIZABLE”（即符合 ACID 中的隔离性），则一条元组在读锁的情况下，对于其他并发事务的读操作是可见的。否则，不可见。如果隔离级别是其他的级别（非“SERIALIZABLE”的其他隔离级别本质上是不符合隔离性的定义的），则根据加锁情况排斥或不排斥其他并发事务的读或写操作，使得数据项对其他事务表现出可见或不可见，反过来说，就是其他并发事务操作过的数据项，对本事务是否可见。判断是否可见，可以依据表 2-4 中的锁在不同隔离级别下释放的时机进行判断。

如果数据库引擎使用了 MVCC 机制和快照隔离技术，则因为一个元组有了多个版本，所以隔离技术使得不同并发事务对同一个数据项的操作相互“隔离”（各自作用在同一个数据项的不同的版本上），此时，除了写 - 写操作外（此句话的背景是 MVCC+ 锁实现并发控制），其他的操作都不互斥，因此在读或写操作这个层面，版本对于事务而言都因不互斥而可见。但是，这不是决定元组可见性的唯一标准，还有三个标准就是数据项的状态，即是否提交（对应 READ COMMITTED、READ UNCOMMITTED 隔离级别）、是否符合事务语义（对应 REPEATABLE READ 隔离级别，语义是在一个事务内前后阶段看到的数据应该是一致的），如果满足当前事务的隔离级别设置，才能说是满足可见性。

综上所述，InnoDB 使用的 ReadView，就是 MVCC 机制中常用的快照隔离，在这种情况下，判断元组对某个事务的可见性，不仅仅指的是 ACID 中的隔离性。

## 2.2 并发控制

在 1.2.5 节介绍了常规的并发控制技术，从检测是否存在冲突的时机的角度看，主要包括乐观的、悲观的以及介于两者之间半乐观的方法；从控制冲突的时机角度看，主要包括：基于锁的、基于时间的、基于提交顺序的、基于串行化图测试验证的等各种方法，另外还有一些其他的方法诸如多版本并发控制技术等。

并发控制的技术有很多，多种技术之间进行组合又创造出更多的并发控制技术。但怎么理解这些技术？为什么多种技术之间可以组合？这些技术之间是否存在内在的关联？

这些是值得探讨的。本节将就着引发数据不一致以及怎样提高并发效率的视角入手，来深入剖析各种并发控制技术是怎么消除数据不一致的、是怎么提高并发效率的。

首先，我们来明确：并发的事务是怎么引发数据不一致的？

其实，我们在 1.1.1 节使用表 1-1 的例子做了一个描述，但这个例子只是一个现象，并不是一个技术答案的表达。

那么，从技术上怎么来回答这个问题呢？回顾 1.2.2 节，我们提出了“可串行化”的概念，并进一步提出了冲突行为、冲突等价、冲突可串行化这几个概念，同时明确说明：

可串行化概念的作用在于保证并发的事务调度方式既能满足数据一致性需求，又能提高并发事务的执行效率。

而在 1.1 节讨论的各种并发问题，如 1.1.3 节的读数据异常问题、1.1.4 节的写数据异常问题、1.1.5 节的快照隔离技术引发的异常问题，以及 1.1.6 节的其他异常问题等，是因为：

- ❑ 存在并发的事务，至少有两个并发的事务（有的异常是至少三个并发事务引发如 1.1.5 节中的三个事务写偏序，提交次序影响了可串行化）。
- ❑ 并发的事务同时操作同样的数据项（读-写、写-读、写-写三种操作，读-读不引发异常）。
- ❑ 事务操作需要实现原子性（要么提交要么中止）导致一个事务存在成功或失败两种可能的状态，失败的事务影响了并发的后提交的事务。

这样，才引发了各种数据异常现象。从技术的角度看，这就是并发的事务引发数据不一致的原因。那么，怎样避免数据不一致的现象呢？答案就是并发执行的事务需要满足“可串行化”。但这个答案是否一定正确呢？其实不然，串行化才能避免数据不一致，可串行化使得事务并发执行，因而满足了上面的几个条件，所以不能保证数据的一致性。所以，需要在“可串行化”这个属性上在增加限制条件，以保证并发的事务在满足“可串行化”时还能保证数据一致性。

这样的限制条件，就是 1.2.2 节中讨论的“可串行化”“可恢复性”“严格性”。

所以，讨论“可串行化”“可恢复性”“严格性”这几个概念之间的关系，是十分必要的。总结 1.2.2<sup>①</sup>节的内容，我们可以得知：

- ❑ 可串行化：从理论上保证了并发事务的调度等价于一个串行调度，避免了数据的不一致现象。
- ❑ 可恢复性：描述了需要避免脏读异常现象，但没有提出实施措施。
- ❑ 避免级联回滚：描述了事务调度机制在“不可恢复性”发生作用时，为了保证数据的一致性不得不降低事务调度机制的效率（所以，这样的事情应该避免）。

① 如果不能理解 2.2 节讨论的内容，可以仔细研读思考 1.2.2 节的内容，这一节是事务并发调度技术的基础。



□ 严格性：提出通过控制并发事务的提交 / 中止操作的顺序，来直接避免脏读现象，这样进一步避免了数据的不一致现象。

以上都满足，才能确保一致性。

其次，我们来讨论：并发的事务调度方法是怎么提高并发度的？

下面我们结合各种并发控制方法，结合本节提出的两个问题，剖析不同的并发控制技术是怎样解决这两个问题的。但是，请注意，2.2 节的各个子节当中讨论的多种并发控制技术，只涉及隔离级别是“Serializable（串行化，或称为可串行化）”的级别，因为只有这个级别才能真正保证数据的一致性。在 2.4 节，我们再结合其他隔离级别讨论并发控制的技术。

首先讨论使用锁方式的并发控制技术。

### 2.2.1 基于锁的并发控制方法

在数据库系统的实现过程中，我们常使用“spinlock”和“latch”来保护共享资源不被并发地写操作同时修改；在编程中，我们引入了对象互斥锁“mutex”的概念，来保证共享数据操作的完整性。这些，都是并发控制的相似技术，这样的技术被称为“锁”，通过事前加锁、事后释放锁，避免了共享资源被不同对象同时修改。所以，封锁技术能够抑制并发，即能避免数据不一致的问题。

但是，如果只是简单的加锁，对于数据库系统而言，一个事务内如果需要读写很多与其他事务共享的数据项，那么这个锁怎么加、加在哪里就是一个问题？

比如说，Linux 下编程实现数据库管理系统，我们定义一个“pthread\_mutexattr\_t data\_mattr;”，对于锁 data\_mattr，如果多个事务同时使用，而事务间完全互斥，则丧失了并发。

那么，我们是否可以定义多个锁？如果定义了多个锁，又如何知道该给哪些事务使用哪些锁呢？

如果把锁加在数据项上，则能解决哪些事务该使用哪些锁的问题。但是，封锁导致不能并发则使得执行效率降低（读-读操作式的并发也被禁止）。而且，需要在一个事务的所有操作完成后才能释放锁，这使得即使多个事务能够并发但执行效率也不高。

多粒度封锁，解决了只使用一种锁导致并发效率低下的问题；两阶段封锁协议，解决了引入多粒度锁之后导致事务调度机制不能保证事务间的并发调度是“可串行化”的。

#### 1. 锁的粒度

锁被分为两种粒度后，即区分为读锁（共享锁）和写锁（排它锁），则并发操作可以被区分为四种：读-读、读-写、写-读、写-写，这样，至少读-读操作的并发可被允许进行，使得在用锁保证数据一致性的情况下，并发能力得以提高。可以得到我们通常所说的“锁的相容性矩阵”。如果进一步把锁的种类再细化，增加意向锁<sup>⊖</sup>等，则可以得到如表 2-4 所示锁的相容性矩阵表，这个表是用于多个事务之间进行事务间加锁时互斥操作使用的（锁的互斥使得并发的的事务间遵循了可串行性）。但是，在实际的编程实战中，还有如表 2-5 所

⊖ 在此我们不再进一步解释意向锁，相关内容，请参见相应的数据库原理书籍。

示锁的升级图，用于在一个事务内对本事务的多个加锁操作进行锁升级的。具体说明，参看表 2-4 和表 2-5。

表 2-4 锁的相容性矩阵表（不同的事务间新锁的申请）

		Granted Mode, 已经授予的锁						
		N	IS	IX	S	SIX	U	X
Requested Mode 正申请的锁	IS	Y	Y	Y	Y	Y		
	IX	Y	Y	Y				
	S	Y	Y		Y			
	SIX	Y	Y					
	U	Y			Y			
	X	Y						

说明：

- 本表表明在同一个数据项上不同事务之间的加锁操作的并发情况。
- “Granted Mode” 表示已经赋予的锁。N 表示不存在锁，IS 共享意向锁，IX 排它意向锁，S 共享锁，SIX 共享排它意向锁，U 更新锁，X 排它锁。
- “Requested Mode” 表示将要申请的锁。
- “Y” 表示可以被授予锁，空白则表示不可以授予新请求的锁。可被授予锁，则意味着事务间的并发操作是被允许的。
- 比如，数据项当前已经被事务 T1 持有 IS 锁，事务 T2 发起 SIX 加锁请求，则事务调度器授予事务 T2 在同一个数据项上持有 SIX 锁。
- 对于 “Granted Mode”，从 N 到 X 这些锁，存在一个排斥的过程，IS 排斥其他并发操作的力度最弱，X 最强，现实中数据库的事务调度器实现实例中在为事务加锁时，就遵循着这样的顺序依序比较，如 Informix、PostgreSQL 等系统。
- 由于可以有多种粒度的锁存在，多粒度封锁协议（multiple-granularity locking protocol）<sup>①</sup>被提出，可以增强并发度，减少锁的开销，本节不再多述，请参阅相关书籍。

表 2-5 锁的合并图（同一个事务内锁的升级）

		Lock Held, 持有的锁					
		IS	IX	S	SIX	U	X
Lock Requested 正申请的锁	IS	IS	IX	S	SIX	U	X
	IX	IX	IX	SIX	SIX	X	X
	S	S	SIX	S	SIX	U	X
	SIX	SIX	SIX	SIX	SIX	X	X
	U	U	X	U	X	U	X
	X	X	X	X	X	X	X

① 《数据库系统概念》，第六版，Abraham Siberschatz等著。

说明：

- 本图表明在同一个事务内对同一个数据项的加锁操作的升级后的结果<sup>①</sup>。
- “Lock Held” 表示已经授予的锁。
- “Lock Requested” 表示将要申请的锁。
- 在同一个数据项上已经授予了 S 锁，如果再申请 IX 锁，则在这个数据项上持有的锁就变为 SIX 锁。
- 在同一个数据项上已经授予了 SIX 锁，如果再申请 U 锁，则在这个数据项上持有的锁就变为 X 锁。其他升级方式不再举例，余者参见表 2-5 所示。

## 2. 两阶段锁

两阶段封锁协议 (two-phase locking protocol, 2PL)，要求每个事务分为两个阶段：

- 增长阶段 (growing phase)：第一个阶段，事务可以获得锁，但不释放锁。
- 缩减阶段 (shrinking phase)：第二个阶段，事务可以释放锁，但不能获得新锁。

两阶段封锁协议可以保证可串行性<sup>②</sup>，但不满足可恢复性，因而不能避免级联回滚；两阶段封锁协议更不满足严格性，所以不能完全保证数据的一致性。

于是，两阶段封锁协议有了新的变种，即：

- 严格两阶段封锁协议 (strict two-phase locking protocol, S2PL)：除了封锁满足两阶段封锁之外，还要求持有的排它锁必须在事务提交后才能释放。这个要求保证未提交事务所写的任何数据在该事务提交之前均以排它方式加锁，从而能够避免级联回滚。如表 2-6 (源自表 1-14) 所示，对于 case 3，事务 T2 的 W(X) 和 Commit 操作，在 S2PL 协议下，已经不可能发生 (事务 T1 在 X 上持有的写锁一直不释放)，所以当事务 T1 执行 Abort 时，级联回滚就不存在。
- 强两阶段封锁协议 (rigorous/strong two-phase locking protocol, SS2PL)：除了封锁满足两阶段之外，还要求事务提交之前不得释放任何锁。这样就满足了严格性的要求 (严格性要求先发生写操作的事务提交或中止的操作优先于其他事务)。
- 在 2.3.2 节，本书对 S2PL 和 SS2PL 进行了更多的比较。
- 主流数据库系统实现的两阶段封锁协议，实际上指的是 SS2PL 而非 2PL。因为只有 SS2PL 才能保证数据的一致性。
- 只使用 SS2PL 实现事务调度的主流数据库系统有 Informix。开源的 PostgreSQL 和 MySQL 不仅使用了 SS2PL，而且还使用 MVCC 来提高事务并发调度的效率。Oracle、DB2、SQL Server 也是基于 SS2PL 和 MVCC 实现的事务并发调度。

① 多种粒度的锁，为了提升并发度，当同一个对象上发生第二次锁申请时，则允许锁进行转换 (lock conversion)，就是锁的升级和降级。升级 (upgrade)：表示从共享到排他的转换，发生在锁增长阶段；降级 (downgrade)：表示从排他到共享转换，发生在锁缩减阶段，多数的数据库不实现降级。

② 可用反证法证明两阶段封锁协议执行的事务过程一定是可串行化的。



表 2-6 可恢复性示例

可恢复				不可恢复	
case 1		case 2		case 3	
T1	T2	T1	T2	T1	T2
R(X)		R(X)		R(X)	
W(X)		W(X)		W(X)	
	R(X)		R(X)		R(X)
	W(X)		W(X)		W(X)
Commit		Abort			Commit
	Commit		Abort	Abort	

### 3. 封锁技术是怎样解决各种读异常的？

对于脏读和不可重复读，封锁技术不会发生这两种读异常现象，原因如表 2-7 所示。

表 2-7 修正后的 SQL 标准定义的三种读数据异常现象与封锁技术的比较表

时间	脏读 P1 ( “Dirty read” )		不可重复读 P2 ( “Non-repeatable read” )		幻象 P3 ( “Phantom” )	
	T1 主事务	T2	T1 主事务	T2	T1 主事务	T2
t0		W(row)- Update	R(row)		R(rows)-WHERE <condition>	
t1	R(row)			W(row)-Update/ Delete		W(rows)-Insert/ Update => <condition>
t2		Abort		Commit	R(rows)-WHERE <condition>	
t3			R(row)			
封锁技术	并发事务 T2 施加写锁成功后，事务 T1 的读操作被阻塞，所以封锁技术中不存在脏读异常		并发事务 T1 施加读锁成功后，事务 T2 的写操作被阻塞，所以封锁技术中不存在不可重复读异常		并发事务 T1 和 T2 操作的不是同一个对象，事务 T1 可以在数据库中存在的数据项上加锁，但事务 T2 可能插入新的数据，这些数据是之前不存在的因而无法用锁排斥，所以幻象异常需要新的解决方式	

对于幻象，需要分为几种情况讨论：

- ❑ 没有索引，使用表或页面级锁：因为锁的粒度大，锁把一些不相关的元组也囊括在锁定的范围内，这使得其他事务的 WHERE 条件不能获取被锁住的表或页面（这相当于锁住了 WHERE 条件指定的数据对象和对象的外延范围，这个外延范围被称为间隙 “gap”），因而可以避免幻象异常。
- ❑ 没有索引，锁的粒度是元组级：因为锁的粒度到了元组一级，即操作多少元组就锁定多少元组对象，所以其他事务的 WHERE 条件能够不受已经加锁的影响，因而可能存在幻象。解决方式是锁的粒度升级为表级或页级。

□ 有索引：在索引上采取“谓词锁 (predicate lock)”来解决。加锁不在一个对象上即不在页面或元组上加锁，而是在“条件”上加锁，逐渐构成一个“条件”锁表来判断新的谓词锁是否可以获得。如 InnoDB<sup>①</sup>中使用“Key-range locking”在索引上锁定一个范围而不是一个条件，在这个范围内的数据项会拒绝删除操作、这个范围也可以拒绝插入和更新操作，所以可以避免幻象读异常现象。注意，这里提到的“谓词锁” (predicate lock) 是一个具有普遍意义的谓词锁“(predicate lock)”的概念，是专门用于避免幻象现象的技术，而幻象产生是因 WHERE 子句中的谓词条件触发，因而取名为谓词锁。但是一些数据库实现，也采用了谓词锁这样的名词，但不能代表本节所提及的谓词锁技术。如 InnoDB 为了支持空间索引引入“谓词锁”、PostgreSQL 为解决 1.1.5 节所提到的快照隔离技术引发的异常而引入“谓词锁”，都是具有普遍意义的“谓词锁 (predicate lock)”的概念的子部分，PostgreSQL 和 InnoDB 中谓词锁的具体详情，参见后续相应章节。

#### 4. 死锁相关

由于使用基于锁的并发控制方法并且锁被分为多种粒度，封锁使得等待产生，则直接导致了死锁现象的产生。如果只有一种锁，并发的事务只有等待该锁被释放后才能有加锁的机会，这样并发的事务处于等待状态。但如果至少存在读写锁，则会发生 R(X)W(Y)-R(Y)W(X) 这样的死锁<sup>②</sup>现象，导致并发事务谁也不能继续执行下去。

那么，死锁是怎么产生的呢？Coffman 等人在 1971 年提出进程在利用可重用性资源时产生死锁的四个必要条件，分别为：

- 互斥使用 (Mutual exclusion)：资源不能被共享，只能由一个进程在同一时刻使用。
- 持有和等待 (Resource holding and waiting)：进程已持有部分资源并等待得到另外的资源，而这些资源又被其它进程所占用还未释放。
- 非抢占分配 (nonpreemption)：已经分配的资源不能从相应的进程中被强行剥夺。
- 部分分配 (partial allocation) 或循环等待 (circular waiting)：至少存在包含两个进程的一个循环链，链中一个进程等待被链中另外一个进程占有的资源。即在一定条件下，若干进程进入了相互无休止地等待所需资源的状态。

而处理死锁，就是消除上面谈到的必要条件，主要有三种方式：死锁预防 (deadlock prevention)、死锁检测 (deadlock detection) + 死锁恢复 (deadlock recovery)、死锁避免 (deadlock avoidance)。这三种方式，可以从讲述数据库系统原理的书籍中找到，本节不再多述。

通常，主流的数据库系统，都实现了自动解除死锁的机制，如 PostgreSQL 使用等待图

① 在 INNODB 中，元组级锁有三种：Record、Gap、Next-KeyLocks。RECORDLOCK 就是锁住某一行元组；而 GAPLOCK 会锁住某一段范围中的元组；NEXT-KEYLOCK 相当前两者加起来的效果。

② 死锁是指两个（或多个）事务相互持有对方期待的锁。



的死锁判断机制，自动检测并预防死锁现象的产生，即把其中将引发死锁的事务中止掉，从而避免死锁现象。Informix 可以设置等待时间，如果等待超时，则自动把引发死锁的事务中止掉，从而避免死锁现象。InnoDB 除了提供等待超时机制外，还使用等待图的算法检测死锁并回滚引发死锁的当前或其他事务。

## 5. 锁的并发度的问题

现在，我们来探讨一下与锁的粒度相关的并发的效率问题。

假设锁不区分为多种粒度，只有一种锁存在，则并发度为 0；而如果有读写锁，只有读-读操作被允许同时进行，并发度为 25%。在图 2-1 中，去掉“Granted Mode”中的 N 列不计，带有“■”的小方格数，除以所有的小方格即 10 除以 36 约等于 27.7778%，这说明细粒度的锁使得并发度再略有提高，这样就增加了并发的效率。之后讨论的 SCO（参见 2.3.4 节）技术，使得读-写操作不阻塞，即只有写-读、写-写操作不能并发，并发度被提高到 50%。之后讨论的 MVCC 结合封锁（参见 2.2.4 节）技术，使得读-写、写-读操作互不阻塞，即只有写-写操作不能并发，并发度被提高到 75%（这就是 MVCC 被广为使用的原因）。

另外，锁的封锁范围，也影响了封锁协议的并发度。

在数据库系统编程实现中，锁可以施加在一个表对象上，也可以施加在一个页面上，还可以施加在一个元组上，分别对应的就是我们常说的表级锁、页级锁、行级锁。如果锁施加到了表或页对象上，则其他事务就不能同时操作这个表对象或者页对象内的其他元组，严重影响了并发度。在磁盘数据库系统中，因为数据的存储是以页面为单位的（与外存存储单位保持一致，都是页面），所以多使用页面作为数据处理的单元，页面被读入数据缓冲区，供上层的数据访问层（Access Method）解析页面结构得到逻辑意义上的元组。而数据缓冲区的组织也是以页面为单位（方便数据映射），所以封锁的单位常以页面为多（尽管主流数据库都提供了行级锁），所以极大地限制了并发事务的并发度。

在内存数据库系统中，因为去除了数据缓冲区，内存索引直接映射了元组，所以主要以元组锁为主，这样，只封锁具体的元组，不殃及不使用的元组，极大地提高了并发事务的并发度。

所以，我们说：锁的封锁范围，也影响了封锁协议的并发度。数据库的事务调度机制，如果使用了基于锁的并发控制方法，就是要在保证可串行化的情况下，尽量减少锁对并发事务的影响，提高并发度，以提高并发执行的事务的执行效率。

## 2.2.2 基于时间戳的并发控制方法

基于时间戳的并发控制技术，是根据事务开始的时间戳值和其他事务的读写操作的时间戳值做比较来决定冲突发生时事务该如何处理。

事务  $T_i$  的时间戳值  $TS(T_i)$  早于事务  $T_j$  的时间戳值  $TS(T_j)$ ，即  $TS(T_i) < TS(T_j)$ ，则并发



调度器必须保证产生的调度等价于事务  $T_i$  出现在事务  $T_j$  之前的某个串行调度<sup>①</sup>。换句话说, 时间戳排序协议保证任何有冲突的 Read 和 Write 操作按时间戳顺序执行。而时间戳的值, 是事务开始时, 由数据库事务管理器直接赋予的 (物理时间值或递增的数值), 这个时间戳值不再发生变化。

但是, 事务执行过程中, 读或写操作的时间戳值会和被操作的数据项紧密相关, 例如:

- $TS(T_i)$  表示事务  $T_i$  发生的时间戳, 即事务开始时刻的时间值。
- $Read(X)-TS(T_i)$  表示事务  $T_i$  在数据项  $X$  上的读动作发生的时刻。
- $Write(X)-TS(T_i)$  表示事务  $T_i$  在数据项  $X$  上的写动作发生的时刻。
- $Commit(X)-TS(T_i)$ , 提交位, 表示事务  $T_i$  在数据项  $X$  上的写动作发生后, 事务  $T_i$  是否提交。
- 读和写这些时刻表明的时间戳值与事务开始的时间戳值是不同的。
- 注意这些值是在数据项  $X$  上存在。即每个数据项上各有一份。

基于时间戳的并发控制技术分为几种情况处理读操作或写操作:

- 假设事务  $T_i$  执行  $Read(X)$  操作 (读-读操作不冲突, 所以不用讨论  $Read(X)-TS(T_i)$  与  $R(X)-TS(non-T_i)$  之间的关系, 如下只讨论读-写、写-读操作, 如图 2-2, 比较的是事务  $T_2$  的开始时间和事务  $T_1$  的写操作的时间):

- 如果  $TS(T_i) < W(X)-TS(non-T_i)$ <sup>②</sup>, 如图 2-2 的 case 1 (事务  $T_2$  的开始时间早于事务  $T_1$  的开始时间) 和 case 2 (事务  $T_2$  的开始时间晚于事务  $T_1$  的开始时间)。事务  $T_i$  需要读入的  $X$  值已被事务  $T_j$  执行了写操作因而被覆盖。因此, Read 操作被拒绝,  $T_i$  被回滚<sup>③</sup> (事务  $T_i$  被中止, 然后被重启及即重新给  $TS(T_i)$  赋予新的时间戳值), 这样解决了写-读冲突<sup>④</sup>。

- 如果  $TS(T_i) \geq W(X)-TS(non-T_i)$ <sup>⑤</sup>, 如图 2-2 的 case 3。事务  $T_i$  开始的时间戳比数据项  $X$  上的写操作晚发生, 则事务  $T_i$  执行 Read 操作更是发生在后, 如果事务  $non-T_i$  被回滚, 则会典型地出现脏读异常, 这是不允许的。所以, 有两种情况需要处理:

- 查看  $Commit(X)-TS(non-T_i)$  的值为 TRUE, 则事务  $T_i$  的读操作被允许执行。
- 否则, 延迟事务  $T_i$  的读操作, 这意味着读操作被阻塞。

①  $TS(T_i) < TS(T_j)$ , 可简单地认为, 基于时间戳并发控制技术, 事务  $T_i$  早于事务  $T_j$  发生, 则事务  $T_i$  的提交/中止也应该早于事务  $T_j$  的提交/中止。即时间戳排序并发控制技术是以事务的开始时间戳值决定可串行性顺序。注意: 这一点, 保证了事务的并发调度满足“可串行性”。

②  $TS(non-T_i)$ , 表示不是事务  $T_i$  的时间戳, 即除了事务  $T_i$  之外的其他事务对应的时间戳。 $W(X)-TS(non-T_i)$  表示不是事务  $T_i$  在数据项上的写操作时间戳; 如果是事务  $T_i$  的写时间戳, 则  $Read(X)-TS(T_i) < W(X)-TS(non-T_i)$  等价于  $Read(X)-TS(T_i) < W(X)-TS(T_i)$ , 这表明是同一个事务之内, 不存在并发冲突所以没有影响。

③ 中止 (撤销) 加重启常称为回滚。

④ 俗称: 过晚读。

⑤ 如果  $TS(T_i) \geq W(X)-TS(T_i)$ , 则是不可能发生的, 即本事务的发生时间戳不可能晚于本事务的写操作时间戳。

○ 本条在描述事务  $T_i$  是发生在其他事务的写操作之前还是之后，其读数据是否被允许。

□ 假设事务  $T_i$  发出  $Write(X)$  操作：

○ 如果  $TS(T_i) < R(X) - TS(non-T_i)$ ，如图 2-3。事务  $T_i$  产生的  $X$  值是先前所需要的值，但事务调度系统已假定该值不应该被产生（因为读操作  $R(X) - TS(non-T_i)$  已经发生过了）。因此， $Write$  操作被拒绝，事务  $T_i$  回滚。这是读 - 写冲突<sup>①</sup>。

○ 如果  $TS(T_i) < W(X) - TS(non-T_i)$ ，如图 2-3。则  $T_i$  产生的  $X$  值是先前所需要的值，但事务调度系统已假定该值不会被产生（因为写操作  $W(X) - TS(non-T_i)$  已经发生过了；即  $W(X) - TS(non-T_i)$  已经被事务调度器认可，而事务  $T_i$  发出的写操作才刚刚进入到事务调度器中被使用到本条规则进行判断）。因此， $Write$  操作被拒绝，事务  $T_i$  回滚。这是写 - 写冲突。

○ 如果  $TS(T_i) < W(X) - TS(non-T_i)$ ，依据上一条描述，事务  $T_i$  被回滚。但是，Thomas 写规则（Thomas write rule）改进了这点，使得事务  $T_i$  的写操作被忽略，即不必执行这个写操作，这样事务  $T_i$  不被回滚，一定程度地提高了并发的效率。所以，Thomas 写规则可以细分写 - 写冲突，在一定程度上提高了并发度。

○ 在事务  $T_i$  内执行  $Write$  操作，并发控制系统将  $Write(X) - TS()$  的时间戳值设为  $TS(T_i)$ 。

○ 本条意在描述事务  $T_i$  是发生在其他事务的读 / 写操作之前，其写数据操作是否被允许。如果发生在其他事务的读 / 写操作之后，则意味着 TO 技术已经决定事务的顺序是  $\langle T_j, T_i \rangle$ ，后发生的事务  $T_i$  内的写操作被允许。

□ 事务  $T_i$  被并发控制机制回滚之后，被赋予新的时间戳并重新启动，自动进入系统（如果不被赋予新的时间戳，则事务  $T_i$  仍然可能与其他事务发生如前述的冲突导致事务  $T_i$  不得不再次回滚，这样就存在事务  $T_i$  得不到提交而处于“提交饿死”状态，被赋予新的时间戳可以有效减少“提交饿死”情况的发生）。

□ 如上可以总结为：

○ 如图 2-2 所示，写 - 读冲突：写 - 读冲突很特别，事务与写作判断。事务早于写操作，读要回滚再重来。事务晚于写操作，读操作不受影响。

○ 如图 2-3 所示，读 - 写冲突：读 - 写冲突很简单，读前写后回滚写。

○ 如图 2-4 所示，写 - 写冲突：写 - 写冲突更简单，新写来临被回滚。

另外，基于时间戳的并发控制协议，不会产生死锁，这是为什么呢？

基于前面的分析，我们可以看到，当有冲突发生的时候，新进入事务调度器被判断为存在冲突的事务立即被回滚，所以不会存在事务形成互为等待的环，因此，死锁现象不会发生。

① 俗称：过晚写。

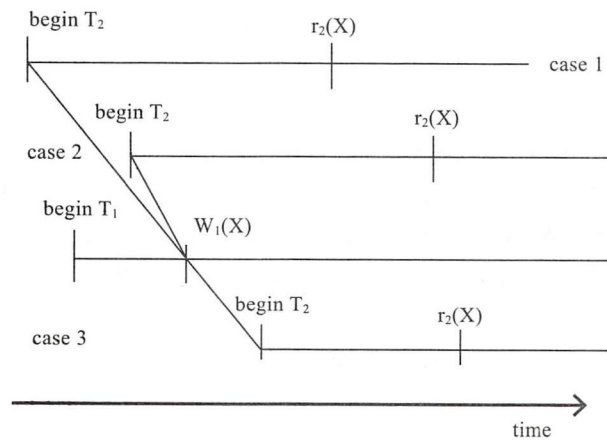


图 2-2 基于时间戳的读操作情况图 (写 - 读冲突)

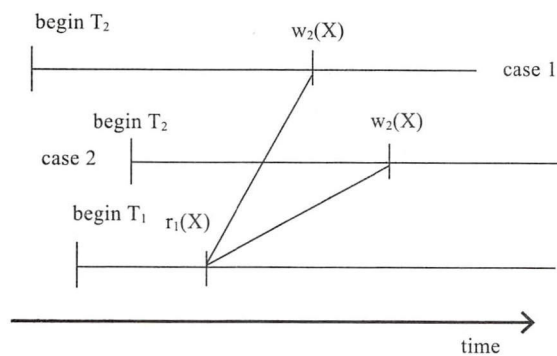


图 2-3 基于时间戳的写操作情况图 (case 1, 读 - 写冲突)

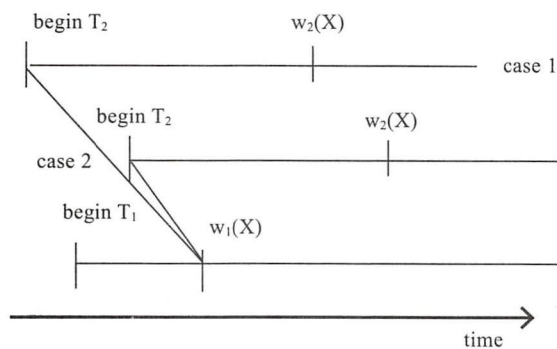


图 2-4 基于时间戳的写操作情况图 (case 2, 写 - 写冲突)



## 1. 基于时间戳的并发调度算法

在实践中，基于时间戳的并发调度算法<sup>①</sup>实现如下，基本原理如上，不再详细描述：

Each transaction ( $T_i$ ) is an ordered list of actions ( $A_{ix}$ ). Before the transaction performs its first action ( $A_{i1}$ ), it is marked with the current timestamp, or any other strictly totally ordered sequence:  $TS(T_i) = NOW()$ . Every transaction is also given an initially empty set of transactions upon which it depends,  $DEP(T_i) = []$ , and an initially empty set of old objects which it updated,  $OLD(T_i) = []$ .

Each object ( $O_j$ ) in the database is given two timestamp fields which are not used other than for concurrency control:  $RTS(O_j)$  is the time at which the value of object was last used by a transaction,  $WTS(O_j)$  is the time at which the value of the object was last updated by a transaction.

For all  $T_i$ :

For each action  $A_{ix}$ :

If  $A_{ix}$  wishes to use the value of  $O_j$ :

If  $WTS(O_j) > TS(T_i)$  then **abort** (a more recent thread has overwritten the value),

Otherwise update the set of dependencies  $DEP(T_i).add(WTS(O_j))$  and set

$RTS(O_j) = \max(RTS(O_j), TS(T_i))$ ;

If  $A_{ix}$  wishes to update the value of  $O_j$ :

If  $RTS(O_j) > TS(T_i)$  then **abort** (a more recent thread is already relying on the old value),

If  $WTS(O_j) > TS(T_i)$  then **skip** (the Thomas Write Rule),

Otherwise store the previous values,  $OLD(T_i).add(O_j, WTS(O_j))$ , set  $WTS(O_j) = TS(T_i)$ , and update the value of  $O_j$ .

While there is a transaction in  $DEP(T_i)$  that has not ended: **wait**

If there is a transaction in  $DEP(T_i)$  that aborted then **abort**

Otherwise: **commit**.

To **abort**:

For each ( $oldO_j, oldWTS(O_j)$ ) in  $OLD(T_i)$

If  $WTS(O_j)$  equals  $TS(T_i)$  then restore  $O_j = oldO_j$  and  $WTS(O_j) = oldWTS(O_j)$

## 2. Thomas 写法则

在基于时间戳的并发控制技术中，有一项改进措施，称为“Thomas 写法则”，其对应的情况如表 2-8 所示。

“Thomas 写法则”有助于改进基于时间戳的并发控制效率，使得特定情况的写操作被“节省”因而提高了执行效率。但是，如果事务 T2 在 t7 时刻不提交而是中止，则会存在一个潜在的问题：被中止的事务 T2 覆盖了数据项 Z 的旧值，Z 的旧值和旧的写时间戳应该被恢复。需要调度器保存数据项 Z 的旧值以便需要恢复时使用，这为事务调度器增加了负担。现实的数据库管理系统中，几乎没有单纯使用基于时间戳排序的并发控制方法的（T0 和其他算法常结合使用）。

① 源自：[https://en.wikipedia.org/wiki/Timestamp-based\\_concurrency\\_control](https://en.wikipedia.org/wiki/Timestamp-based_concurrency_control)

表 2-8 Thomas write rule

时间	T1	T2	说明
t0	Begin		
t1		Begin	事务 T1 先于事务 T2 开始, 提交次序应该是事务 T1 优先
t2		R(X)	
t3	R(Y)		
t4		W(Z)	事务 T2 写数据项 Z
t5	W(Z)		事务 T1 也写数据项 Z, 这个写操作可以被忽略。 因为后发生的事务的写操作会覆盖之前事务的写操作产生的值
t6	Commit		事务 T1 先提交, 符合时间戳排序协议
t7		Commit	

### 2.2.3 基于有效性检查的并发控制方法

在上一节, 我们讨论了基于时间戳排序的并发控制技术, 基于有效性检查的并发控制技术和基于时间戳的技术有很大相似之处, 所以本节我们换一种讨论方式, 先讨论这两种技术的相同点和差异, 借助时间戳排序技术理解有效性检查的并发控制技术。如表 2-9 所示。

表 2-9 基于时间戳排序的并发控制技术与基于有效性检查的并发控制技术的比较表

	基于时间戳排序	基于有效性检查
如何决定可串行性	以事务的开始时间戳值决定可串行性	以事务的有效性验证时间戳值决定可串行性
并发事务间是否存在先后顺序	存在。事务的开始时间戳值小者居先	存在。事务的有效性验证时间戳值小者居先
冲突行为	存在, 包括: 读-写、写-读、写-写	有效性检查不通过则意味着存在冲突
与基于时间戳排序并发控制技术的关系	就是自己	基于时间戳排序协议, 事务的提交顺序只是不依据事务的开始时间戳, 而是依据有效性检查时间戳
基于有效性检查并发控制技术的三个阶段是特有的吗?	基于时间戳排序协议不需要这三个阶段	基于有效性检查并发控制协议需要这三个阶段, 是自身特有的协议
为什么基于有效性检查并发控制技术引入了一个有效性检查阶段?	没有“有效性检查阶段”是因为本协议在读写操作的过程中就判断了是否存在冲突, 如果有冲突立刻进行了处理(采取 Abort 操作达到回滚事务的目的)	首先, 在读阶段(如下文), 数据被复制到事务的局部变量中, 写操作是对局部变量进行的, 在这个阶段内, 这样做不会引发冲突; 所以下一阶段即有效性检查阶段直接判断是否与其他事务存在冲突; 如果没有冲突, 直接进入第三阶段写阶段

基于有效性检查的并发控制技术(validation protocol), 对于每一个事务, 在其生命周期中被划分为两个或三个阶段, 每个事务必须按如下顺序依次执行:

□ 读阶段(read phase): 对于事务  $T_i$ , 涉及的数据项被读入到事务  $T_i$  的局部变量中,



所有的写操作都是对局部变量进行修改，并不对数据库中的数据项进行真正更新。因此，没有读-写、写-读、写-写操作产生冲突。即不用延迟冲突操作，因此，在读写阶段事务的并发度会很高。同时也表明，事务在“乐观”地执行着，冲突只在读和写操作完成后才被判断是否存在。所以这是一种乐观的并发控制技术。

- 有效性检查阶段 (validation phase): 根据有效性检查规则，对事务  $T_i$  进行有效性测试 (即判断事务  $T_i$  的 Write 操作是否违反可串行性)，如果违反可串行性即事务有效性测试失败，则终止这个事务。
- 写阶段 (write phase): 如果事务  $T_i$  的有效性检查通过，则把事务  $T_i$  的被写过的局部变量的值复制到数据库中。只读的事务不需要本阶段判断。

其中，有效性检查阶段提到的“有效性检查规则”是指：

- 首先，标示事务  $T_i$  的各个阶段是何时进行的。
  - Start( $T_i$ ): 事务  $T_i$  开始执行时间。
  - Validation( $T_i$ ): 开始有效性检查的时间。
  - Finish( $T_i$ ): 事务  $T_i$  完成写阶段的时间。
- 其次，利用时间戳的值，通过时间戳排序并发控制技术决定可串行性。
  - 对比 2.2.2 节，可以推知  $TS(T_i) = \text{Validation}(T_i)$ ，即利用有效性检查的时间作为事务提交顺序的依据 (基于时间戳排序并发控制技术的  $TS(T_i)$  是事务开始的执行时间)。
  - 如果  $TS(T_i) < TS(T_j)$ ，则产生的任何调度必须等价于事务  $T_i$  出现在  $T_j$  之前的某个串行调度。
  - 必须保证  $\text{Finish}(T_i) < \text{Start}(T_j)$ ，事务  $T_i$  在事务  $T_j$  开始之前完成执行，保证了可串行性 (即用一定是串行的次序确保了可串行性，但是没有并发)。
  - 进一步，即能提高并行度又能保证可串行性的一个改进方式是：保证事务  $T_i$  所写的数据项与事务  $T_j$  的所读的数据项不相交。即： $\text{Start}(T_j) < \text{Finish}(T_i) < \text{Validation}(T_j)$ ，事务  $T_i$  的写阶段必须在事务  $T_j$  的有效性检查阶段之前完成。可串行性得到保证，是因为：这个条件保证  $T_j$  与  $T_i$  的写不重叠，即  $T_i$  的写不影响  $T_j$  的读，而当不满足可串行性时，事务  $T_j$  的有效性检查阶段可以确保冲突发生后事务  $T_j$  可被回滚。

## 2.2.4 基于 MVCC 的并发控制方法

多版本并发控制 (Multiversion concurrency control, MVCC) 技术，核心思想是：

- 当事务  $T_i$  执行一个读操作，并发控制器选择一个版本进行读取，这个版本的获取，依赖于事务  $T_i$  所能读取的数据的上下文；这个上下文 (有个通俗名称，叫做事务的快照，快照要和多版本相配合) 是事务  $T_i$  在数据库系统里的当前并发执行的诸多事务的状态的一份拷贝，其中的信息能够帮助判断：获取到的元组是否对本事务是可





读取的<sup>①</sup>?

- ❑ 多版本并发控制技术不是一个可独立使用的事务并发控制技术，而是需要基于其他并发控制技术，如基于时间戳的称为“多版本时间戳排序机制（multiversion timestamp-ordering scheme）”，基于两阶段封锁协议的称为“多版本两阶段封锁协议（multiversion two-phase locking protocol）”。如下分别讨论这两种并发控制技术，并如表 2-10 所示进行比较。

表 2-10 两种 MVCC 并发控制技术比较表

比较项	多版本时间戳排序机制	多版本两阶段封锁协议
可串行化的方式	通过开始事务的时间戳值来排序事务的提交顺序以及本协议确定的规则来确保可串行化	通过两阶段封锁机制和严格性确定事务的提交顺序来确保可串行化
读请求不失败不等待	是	是
读-写冲突的解决方式	写操作被撤销	写操作能够执行，生成一个新的版本
写-读冲突的解决方式	读不被阻塞	读不被阻塞
写-写冲突的解决方式	取决于事务的时间戳，可能被回滚，也可能被允许执行	排它锁保证后发生的写操作被阻塞
产生潜在的磁盘访问	读取数据项要更新 R-timestamp(Xi) 字段，一读一更新共两个 IO	—
是否区分只读事务和更新事务	不区分	区分
适用范围	大多数事务为只读或很少有并发事务更新相同数据项	并发冲突高的情况（延迟一些操作避免了事务回滚带来的更大开销）

### 1. 多版本时间戳排序机制

- ❑ 首先，数据库系统在事务开始前赋予一个时间戳，记为  $TS(T_i)$ ，这个时间戳则决定了并发的事务的调度顺序。
- ❑ 其次，对于每个数据项  $X$ ，多版本体现在： $X$  有一个版本序列  $\langle X_1, X_2, \dots, X_n \rangle$ ，其中，每个版本  $X_i$  包括三个字段，分别是：
- $X_i = \text{value}$ ，value 是数据项  $X$  的第  $i$  个版本的值，每个版本是由一个写操作生成的。
  - $W\text{-timestamp}(X_i)$  是创建  $X_i$  这个版本的事务的时间戳（不是当前时间戳值），即表明此数据项是被谁在什么时候创建的。
  - $R\text{-timestamp}(X_i)$  是所有成功读取  $X_i$  这个版本的事务的时间戳。
- ❑ 再次，多版本时间戳排序机制通过如下规则，保证可串行性：
- 如果事务  $T_i$  执行 Read 操作或 Write 操作，假设  $X_m$  表示  $X$  满足如下条件的版本，其写时间戳是小于或等于  $TS(T_i)$  的最大写时间戳（确保了在所有版本中找到一个“最近版本”）。
  - 如果事务  $T_i$  执行读操作  $\text{Read}(X)$ ，返回给事务  $T_i$  的值为  $X_m$ 。读永远不会被阻塞。

① 这个叫做“元组可见性判断”。PostgreSQL 和 MySQL 等数据库都实现了 MVCC 都需要判断元组的可见性。



○ 如果事务  $T_i$  执行写操作  $Write(X)$ :

○ 且如果  $TS(T_i) < R\text{-timestamp}(X_m)$ , 则中止事务  $T_i$ , 这表明即将执行的这个写操作之后的时间上已经发生过了一个读操作, 如果允许写操作成功, 则可能发生不可重复读异常现象。这是写-读冲突, 事务  $T_i$  被中止。注意这一点, 发生此种情况的时候, 是因为事务  $T_i$  的写操作本来在物理时间上早于对  $X_m$  这个版本的其他事务的读操作, 但因为并发执行是无序的, 导致调度器在判断“ $TS(T_i) < R\text{-timestamp}(X_m)$ ”时刻之前,  $R\text{-timestamp}(X_m)$  已经发生了。所以才能有“ $TS(T_i) < R\text{-timestamp}(X_m)$ ”这种现象发生。

○ 且如果  $TS(T_i) = W\text{-timestamp}(X_m)$ , 则系统更新事务  $T_i$  的值  $X_m$  为新值, 这表明本事务多次写过同一个数据项, 新值覆盖旧值。

○ 且如果  $TS(T_i) > W\text{-timestamp}(X_m)$ , 则系统为事务  $T_i$  的数据项  $X$  创建一个新值。这说明后发生的事务才创建新的版本。这是写-写冲突, 导致产生新版本。

## 2. 多版本两阶段封锁协议

□ 首先, 每个数据项  $X$ , 其多版本体现在:  $X$  有一个版本序列  $\langle X_1, X_2, \dots, X_n \rangle$ , 其中, 每个版本  $X_i$  包括一个时间戳, 多数数据库中, 这个时间戳对应的是唯一的一个事务标识, 叫做事务号。

□ 事务号通常是一个递增的数字。

□ 其次, 事务分为两种类型, 一种是只读事务<sup>①</sup>, 另外一种是更新事务。这意味着需要事先知道事务的读写请求、进而告知事务管理器, 事务管理器才能区分事务的类型然后根据类型做出一定的优化(多数多版本两阶段封锁协议的事务管理器都采取了一定措施为只读事务做了优化)。

□ 只读事务开始获得事务号, 根据事务号读取事务号对应的数据项的版本, 只读事务在整个生命周期内只使用这个版本的数据。

□ 更新事务的操作可以细分为:

○ 更新事务中的读操作, 如果能获取该数据项的共享锁, 读取该数据项的最新版本的值。

○ 更新事务中的写操作:

■ 如果能获得该数据项的排它锁, 则为该数据项创建一个版本。刚创建的时候新版本的时间戳值为无穷大, 致使其他并发事务不能读取到此尚未提交的数据项的版本。事务提交, 此版本上的时间戳值+1, 表示此后其他事务可以读写此版本的数据。

■ 不能获得排它锁时, 表明有其他事务准备或已经写了数据但没有结束即锁没有被释放, 本更新事务只能等待。

① 有的数据库系统号称支持“只读事务”, 就是因为使用了多版本两阶段封锁协议。如PostgreSQL的事务并发控制技术采取的就是多版本两阶段封锁协议, 然后对只读事务做了优化。



在表 2-10 中,我们提到的读操作,对应着 SQL 语句的 SELECT 操作,这样的操作,在快照隔离技术中,可以做到读不加锁。但是,“读”操作的另外一层含义,是“获取”,即某 SQL 语句需要先获取到数据,然后才能操作数据,这样的获取,又很多时候,也被称为“读”操作,这算是一种不严谨的说法。例如在 MVCC 并发控制方法中,不严谨的说法里“读”操作可以细分为两种情况:

- ❑ 快照读 (snapshot read): 读取的是元组的可见版本 (可见的含义是获取在快照允许的事务范围内的版本,这样的版本可能是历史版本),不用加锁。如 InnoDB 使用 UNDO 日志帮助获取快照范围内的历史版本以支持简单的 SELECT 查询操作。
- ❑ 当前读 (current read): 读取的是记录的最新版本,被当前读返回的元组会加上锁,保证其他事务不会再并发修改这条元组。如 InnoDB 中把 UPDATE、INSERT、DELETE 操作视为特殊操作,这样的操作需要读取数据,属于当前读,也属于前面谈及的不严谨的“读”操作。

DML 和 DQL 既然都要操作数据,所以都要“读取”数据。但是 DQL (普通查询语句,不是指带有 FOR UPDATE 等显式加锁的语句)只是读出数据显示而已,但 DML 读出后却要进行写操作所以不只是读,目的不同,自然应该施加的锁是不同的。所以从这个角度上看,快照读根本不用加锁,当前读一定要加锁,所以根本不必区分快照读和当前读这两种读,只是简单认为快照读基于多版本所以不必加锁,DML 属于写操作,自然要加锁处理。所以笔者认为,区分出当前读这个名词是不必要的。

### 2.2.5 基于 MVCC 的可串行化快照隔离并发控制方法

Serializable Snapshot Isolation,可串行化的快照隔离,简称 SSI。

首先,这种技术,基于 MVCC 中的多版本,也基于快照隔离的思想。

其次,为了解决快照隔离的写偏序异常问题,引入了本项改进技术。

第三,SSI 因基于 SI (快照隔离)所以整体流程与 SI 相同,只是增加了一些“book-keeping”记录下事务的一些信息以便动态地检测是否有写偏序现象的发生(工程实现中是检测可能有写偏序发生而不一定是写偏序发生,所以存在误判的可能,这么做是为了提高检测的效率),如果有,则回滚引发写偏序异常的事务。

可序列化的快照隔离技术的主要内容,参见论文《Serializable Isolation for Snapshot Databases》。下面我们分三个方面,来探讨 SSI 的具体技术实现。

#### 1. 理论基础

检测写偏序的理论基础依赖于两篇论文:

- ❑《Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions》: 定义了“读写依赖 (rw-dependency)”,通过读写依赖表明不可串行化必然是一个环内存在两个读写边于多版本可串行化图 (the multiversion serialization graph, 简称 MVSG) 中。





- ❑《Making snapshot isolation serializable》：定义了“读写依赖 (rw-dependency)”的扩展形式，表明前述的两条边相邻且每条边的两个端点代表着不同的活动状态下的事务。
- ❑上述两篇论文，实则在表明并发事务之间的读写操作是怎么造成事务的“冲突行为”（1.2.2 节）之间的逻辑关系的。
- ❑利用上述两篇论文，可以构造出并发事务如何形成一个环，环存在表明事务之间相互依赖以至于形成了写偏序异常，所以解决写偏序异常的方式就是打破环，回滚使环形成的新加入事务，从而破解写偏序异常现象。

## 2. 三种依赖关系

前面概略地讲述了解决写偏序的理论基础，其中提到了“读写依赖 (rw-dependency)”，接下来我们来细化三种依赖关系，这三种依赖关系定义自论文《Generalized isolation level definitions》，具体如下，图 2-5 是原文对三种依赖关系的描述：

- ❑“写-读依赖 (wr-dependency)”：事务 T1 写数据项 X 的一个版本，事务 T2 读这个版本，意味着事务 T1 先于事务 T2 执行，所以可以把事务 T1 作为起点、T2 作为终点，画一条从 T1 到 T2 的边，即对同一个版本的写操作到读操作的边。
- ❑“写-写依赖 (ww-dependency)”：事务 T1 写数据项 X 的一个版本，事务 T2 使用一个新版本替换这个版本，意味着事务 T1 需要先于事务 T2 执行完毕（T1 提交完成），所以可以把事务 T1 作为起点、T2 作为终点，画一条从 T1 到 T2 的边，即对同一个对象的写操作到写操作的边。
- ❑“读-写依赖 (rw-dependency，又称为 anti-dependency 或 rw-conflicts)”：事务 T1 写数据项 X 的一个版本，事务 T2 读这个对象之前的版本，意味着事务 T1 后于事务 T2 执行，所以可以把事务 T1 作为起点、T2 作为终点，画一条从 T1 到 T2 的边（虚线绘制，实际上应当是反向倚赖即写倚赖于读），即对同一个对象的写操作到读操作的边。
- ❑在第一章的图 1-1 和 1-2 中，我们给出过三种依赖关系的图，这些图中边的含义，如上描述。
- ❑如上的几种依赖关系，可以画出并发事务间的一个依赖图，如果存在了环，则表明互为依赖（语义上表明写偏序异常发生），所以解决问题的方式就是打破环。
- ❑所以，从理论上写偏序的问题被抽象为在并发事务之间绘制依赖图，如果存在环则意味着写偏序发生，所以解决写偏序采用的方式是打破环即可。
- ❑但是，工程实践当中的解决方式却不完全是这样，论文《Serializable Isolation for Snapshot Databases》提出的算法是在即将形成环之前，通过回滚某个事务破坏形成环的条件，从避免环形成写偏序异常。
- ❑如下展示的算法，是当发现有两个相邻的 rw-dependency 时则回滚当前事务，这样在环形城前即破获了环形成的条件，从而实现了可串行化快照隔离 SSI，避免了写偏序异常。



Conflicts Name	Description ( $T_j$ conflicts on $T_i$ )	Notation in DSG
Directly write-depends	$T_i$ installs $x_i$ and $T_j$ installs $x$ 's next version	$T_i \xrightarrow{ww} T_j$
Directly read-depends	$T_i$ installs $x_i$ , $T_j$ reads $x_i$ or $T_j$ performs a predicate-based read, $x_i$ changes the matches of $T_j$ 's read, and $x_i$ is the same or an earlier version of $x$ in $T_j$ 's read	$T_i \xrightarrow{wr} T_j$
Directly anti-depends	$T_i$ reads $x_h$ and $T_j$ installs $x$ 's next version or $T_i$ performs a predicate-based read and $T_j$ overwrites this read	$T_i \xrightarrow{rw} T_j$

Figure 2. Definitions of direct conflicts between transactions.

图 2-5 《Generalized isolation level definitions》定义的三种依赖关系

### 3. 算法实现

每个事务对象上，有两个 boolean 值的元素，分别表示：

- $T.inConflict$ : TRUE 值表示有一个 rw-dependency 从别的并发事务指向自己。
- $T.outConflict$ : TRUE 值表示有一个 rw-dependency 从自己指向别的并发事务。
- $T.inConflict$  和  $T.outConflict$  都为 TRUE 时，则意味着有相邻的两个 rw-dependency，这就是一个“可能的”不可串行化的快照隔离。

引入一个新的锁模式称为“SIREAD”锁：

- 首先，多版本的快照隔离本质不是基于锁的并发控制技术，所以这里不应该和锁扯上关系。
- 其次，MVCC 技术通常配合其他并发控制技术，以提高并发度。如 2.2.4 节，我们听到过基于时间戳的“多版本时间戳排序机制 (multiversion timestamp-ordering scheme)”，基于两阶段封锁协议“多版本两阶段封锁协议 (multiversion two-phase locking protocol)”。因多数数据库管理系统在编码实现的时候是基于 SS2PL 技术的，所以 SSI 技术基于两阶段封锁协议，提出了“SIREAD”锁。换句话说： $SSI=SS2PL+MVCC (+SI) +SIREAD$  锁。
- 第三，“SIREAD”锁表示一个 SI 事务 (使用快照隔离技术的事务) 在数据项上读取了一个版本。“SIREAD”锁不会阻塞任何锁 (与任何锁都相容)，所以“SIREAD”锁看起来更像是一个标志而不是“锁”。“SIREAD”锁是加在数据项对象上的，不是施加在某个版本上。
- 第四，如果一个数据项的某个版本上存在有 SIREAD 锁和 WRITE 锁，表示存在 rw-dependency 关系，因此持有这些锁的某个事务可以设置其  $inConflict$  和  $outConflict$  值。这样就把锁和 SSI 技术关联起来。
- 第五，SSI 算法主要体现在四种操作上，分别是：事务开始、读操作、写操作、事务提交。这四种操作的具体算法如下：





## (1) 在事务 T 开始的时候:

```
modified begin(T):
existing SI code for begin(T) //设置事务T的inConflict和outConflict的初始值为FALSE
set T.inConflict = T.outConflict = false
```

## (2) 当事务 T 发生读操作时:

```
modified read(T, x):
get lock(key=x, owner=T, mode=SIREAD) //在数据项x上施加SIREAD锁,并设定属主为当前事务
if there is a WRITE lock(wl) on x //检查数据项x,当有写锁存在时,则事务T有rw-
dependency指向写锁的属主事务:所以设置其inConflict值为TRUE,设置设置事务T的outConflict值为TRUE
set wl.owner.inConflict = true
set T.outConflict = true
existing SI code for read(T, x) //SI的算法实现,即SSI依旧要使用SI的算法
for each version (xNew) of x //检查数据项x上的每一个新版本xNew⊖
that is newer than what T read:
if xNew.creator is committed //如果xNew的创建者已经提交
and xNew.creator.outConflict: //且其outConflict值为TRUE
abort(T) //意味着有两个相邻的rw-dependency存在,所以需要回滚本事务T
return UNSAFE_ERROR
//否则,这是一个rw-dependency关系,此关系是事务T指向数据项x的新版本的创建者事务的
set xNew.creator.inConflict = true //可以正常地为xNew的创建者事务设置inConflict值为TRUE
set T.outConflict = true //为本事务T的outConflict设置值为TRUE
```

## 说明:

读操作发生时,“意味着有两个相邻的 rw-dependency 存在”,是表明事务 T 处于相邻的存在 rw-dependency 关系的事务的尾端。即:另外的一个事务 → xNew.creator → T。

## (3) 当事务 T 发生写操作时:

```
modified write(T, x, xNew):
get lock(key=x, locker=T, mode=WRITE) //在数据项x上施加WRITE锁,并设定属主为当前事务
if there is a SIREAD lock(rl) on x //检查数据项x,当有SIREAD锁存在时
with rl.owner is running //且施加SIREAD锁的事务rl.owner还在运行(没有提交或回滚)
or commit(rl.owner) > begin(T): //或者rl.owner的提交时间晚于事务T的开始时间
if rl.owner is committed//意味着有两个相邻的rw-dependency存在
and rl.owner.inConflict:
abort(T) //所以需要回滚本事务T
return UNSAFE_ERROR
//否则,这是一个rw-dependency关系,由创建SIREAD锁的事务指向本事务T
set rl.owner.outConflict = true
set T.inConflict = true
existing SI code for write(T, x, xNew) //SI算法实现,即SSI依旧要使用SI的算法
# do not get WRITE lock again
```

⊖ 事务T不应该读到(在事务T开始时没有完成提交的事务)但在事务T结束前已经提交的事务才会生成新的T读不到的版本。





说明：

读操作发生时，“意味着有两个相邻的 rw-dependency 存在”，是表明事务 T 处于相邻的存在 rw-dependency 关系的事务的头端。即：T→rl.owner→ 另外的一个事务。

(4) 当事务 T 提交时：

```
modified commit(T):
if T.inConflict and T.outConflict: //意味着事务T处于两个相邻的rw-dependency的中间
abort(T) //存在两个相邻的rw-dependency, 所以要回滚
return UNSAFE_ERROR
existing SI code for commit(T) //SI算法实现, 即SSI依旧要使用SI的算法
# release WRITE locks held by T
# but do not release SIREAD locks //SIREAD锁不被释放
```

说明：

SIREAD 锁不被释放，因为提交后的事务也会影响其他正在运行的事务，示例如第一章中表 1-4 中的“三个事务写偏序”的情况和图 1-2 的“三个事务引发的异常现象”，这里不再赘述。

## 2.2.6 再深入探讨三种读数据异常现象

Q6：表 1-11 中，不可重复读现象中的事务 T2 在 t2 时刻执行“提交 Commit”或不执行“Commit”会有什么差别吗？或者对于幻象现象，事务 T2 在 t2 时刻没有执行“Commit”，这一点与不可重复读有差别吗？

答：

对于“不可重复读异常”现象，如果并发控制技术使用的是两阶段封锁协议，事务 T2 在 t1 时刻不会被阻塞，所以事务 T2 能够在 t2 时刻提交。而事务 T1 在 t3 时刻读取数据的操作如果被允许，则会发生“不可重复读异常”现象，所以“可重复读”隔离级别下事务 T1 在 t3 时刻读取数据的操作将失败（即用事务 T1 的失败来避免“不可重复读异常”现象发生）<sup>①</sup>。但是如果使用了多版本并发控制协议，事务 T1 读取的数据是旧版本的数据不受事务 T2 在 t1 时刻写的新版本的数据影响，因而能避免“不可重复读异常”现象。

对于“幻象异常”现象，在 t1 时刻，因为事务 T2 已经在相同数据项上施加了写锁（更新锁或排它锁），事务 T1 被阻塞不能在 t2 时刻执行读操作。但如果基于多版本的封锁并发控制技术（参见 2.2.4 节），t2 时刻事务 T1 的这个读操作不会被阻塞，通常情况下，不被阻塞的读操作会出现“不可重复读异常”现象，但是因为使用了多版本，即事务 T1 读取的数据是旧版本的数据不受事务 T2 在 t1 时刻写的新版本的数据影响，因而能避免“幻象异常”现象。

① 有的系统在此种情况下，通过延迟事务 T2 的提交操作，来避免事务 T1 的失败。这使得在 SS2PL 机制下，读操作阻塞了写操作，并发度极低。



在商业或开源的数据库系统实现中，对于幻象的处理，还需要细分上一段文字所述的情况：

- 不存在可利用的索引查询数据：事务 T1 在 t0 时刻的读操作做全表扫描：
  - 锁粒度是表级或页级：事务 T1 申请加读锁，事务 T2 在 t1 时刻写数据申请到了表级或页级写锁（表或页对象上的锁进行了锁升级的动作）：
    - 事务 T2 在 t1 时刻写数据成功，如果并发控制使用不同的协议，则实际结果不同：
      - SS2PL 并发控制协议：事务 T1 在 t2 时刻的读操作则不被允许（写锁排斥了读锁）。
      - MVCC+SS2PL 并发控制协议：事务 T1 在 t2 时刻的读操作则是否被允许，取决于所 T1 读取的数据和 T2 写的的数据是不是在一个页码上，如果在一个页面上，则读锁依然被写锁排斥，所以不允许事务 T1 在 t2 时刻执行读操作（这是通过截止读操作以避免幻象异常现象的方式）。
    - 锁粒度是元组级：事务 T1 申请加读锁，事务 T2 在 t1 时刻写数据申请到了元组级写锁（元组对象上的锁进行了锁升级的动作）：
      - 事务 T2 在 t1 时刻写数据成功，如果并发控制使用不同的协议，则实际结果不同：
        - SS2PL 并发控制协议：事务 T1 在 t2 时刻的读操作则不被允许（写锁排斥了读锁）。
        - MVCC+SS2PL 并发控制协议：事务 T1 在 t2 时刻的读操作则被允许，读取旧版本的值，这种情况下才能显示出多版本的优势。多版本和 SS2PL 并发控制技术协同保证了不出现幻象异常现象（这是保证满足读操作的情况下不出现幻象异常现象）。
  - 存在可以利用的索引查询数据：如果有索引可以使用，幻象异常现象的避免，通常是在索引上采取“谓词锁（predicate lock）”来解决的。加锁不在一个对象上即不在页面或元组上加锁，而是在“条件”上加锁，逐渐构成一个“条件”锁表来判断新的谓词锁是否可以获得。如 InnoDB 中使用“Key-range locking”在索引上锁定一个范围而不是一个条件，在这个范围内的数据项会拒绝删除操作、这个范围也可以拒绝插入和更新操作，因而可以避免幻象读异常现象。
  - 分析以上内容，数据库事务管理器要保证不出现读异常现象，需要考虑并发控制协议、如使用 SS2PL 协议再考虑有无索引、锁的粒度、是否存在 MVCC 技术、隔离级别等多种纬度，组合起来异常复杂，这导致了对事务处理技术的掌握很困难。
  - 在第六章，我们将以 PostgreSQL 的源码为例，实例探讨采取 MVCC+SS2PL 并发控制技术究竟是如何解决各种读异常现象的。

如上讨论了在两阶段封锁技术和多版本技术结合使用的情况下，事务管理器是怎么解决这些异常现象带来的数据不一致的问题的。

这三种读异常现象，在其他的并发控制技术下，有着不同的应对和处理方式，读者可



以结合其他并发控制技术思考不同的并发控制技术是如何解决这些异常现象带来的数据不一致的问题的，本书不再赘述。

Q7：表 2-5 中，row 是一个什么级别的数据？

答：

对于行存数据库，row 可以是以行为单位的行或行中的某些列。其含义是一个“数据项”。

Q8：表 2-5 中，幻象一定要带有 WHERE 条件子句吗？

答：

不是。没有 WHERE 相当于 WHERE<TRUE> 即表示所有行满足条件。

## 2.3 并发控制技术的比较

并发控制技术，解决的问题是：在多个事务并发执行的情况下，在必须保证数据一致性的情况下，如何提高并发效率。

为了保证数据的一致性，需要可串行化、可恢复性、严格性等属性来约束并发事务的调度机制，即需要确立并发时怎么保证数据的一致性。

而事务的并发调度，主要包括两个方面和一个效果：

- 一是事务的读操作和写操作等之间的关系、即如何解决（阻塞还是允许）冲突行为（读-写、写-读、写-写）。
- 二是并发事务间的提交 / 中止的顺序。
- 一个效果是指每一种并发控制技术所讨论的就是在上面的两条规则限定下，并发度如何被提高。即并发执行的效率问题。

在 2.2 节讨论的各种并发控制技术，就是着重于讨论上述两个方面和一个效果的。请回顾上一节内容，体会并发控制技术是怎么从上述二个方面入手，既保证数据的一致性又提高并发效率的。

### 2.3.1 并发控制技术整体比较

在 2.2 节我们讲述了多种并发控制技术，对于每一种并发技术，我们重点讲述其怎么能够符合可串行化等事务属性，意在表明这种并发控制技术是怎么保证数据一致性的；重点讨论每种并发控制技术的协议规则，意在表明这种控制技术是怎么提高并发度的。前者在讲正确性，后者在讲效率。

在本节，我们对 2.2 节的内容做总结，分别对各种并发控制技术的特点做比较（如表 2-11 所示）、对各种并发控制技术做整体比较（如表 2-12 所示）、对各种并发控制技术的冲突处理方式做比较（如表 2-13 所示）。





表 2-11 并发控制技术特点比较表

类别	子类别	保证可串行化	保证可恢复性	保证无级联性	保证冲突可串行化	存在死锁	饿死长事务	冲突事务等待或回滚	版本删除
时间戳	时间戳排序协议	是	不 <sup>①</sup>	不 <sup>②</sup>	是	不	可能	回滚	—
	Thomas 写规则	是	不	不	是	不	可能	回滚	—
	多版本时间戳排序 <sup>③</sup>	是	不	不	是	不	可能	回滚	需要 <sup>④</sup>
锁	两阶段封锁	是	不	不	是	是	不可能	等待	—
	严格两阶段封锁	是	是	是	是	是	不可能	等待	—
	强两阶段封锁	是	是	是	是	是	不可能	等待	—
	基于图的封锁协议/树形协议 <sup>⑤</sup>	是	不	不	是	不	不可能	等待	—
	多粒度封锁协议	是	是	是	是	是 <sup>⑥</sup>	不可能	等待	—
	多版本两阶段封锁 <sup>⑦</sup>	是	是	是	是	是	不可能	等待	需要 <sup>⑧</sup>
有效性检查	有效性检查协议	是	是	是	是	不	可能 <sup>⑨</sup>	重启 <sup>⑩</sup>	—

① 协议改进后（在事务的末尾执行写操作，在写操作正在执行时，其他事务不允许访问已经写过的数据项）可保证可恢复性。

② 协议改进后（在事务的末尾执行写操作，在写操作正在执行时，其他事务不允许访问已经写过的数据项）可保证无级联性。

③ 写操作可能引起事务的回滚。

④ 某个版本的写时间戳小于最老的事务的时间戳，则可被删除。

⑤ 与两阶段封锁协议的差别在于，树形协议需要事先知道访问数据项的顺序，如适用于执行存储过程。

⑥ 有多种技术可以改进多粒度封锁协议，使得减少死锁发生的频度，或者消除死锁（H.F.Korth “Deadlock Freedom Using Edge Locks”）。

⑦ 写操作可能导致封锁等待或死锁。

⑧ 某个版本的写时间戳小于等于最老的只读事务的时间戳，则可被删除。

⑨ 一系列冲突的短事务会引起长事务反复重启。

⑩ 事务回滚后又被自动重启。重启是有效性检查协议明显区别于其他并发控制技术的特点之一。

表 2-12 并发控制技术的整体比较

并发控制技术	存储的利用率	性能
封锁技术	锁表空间与被封锁元素个数成正比	推迟了事务但避免了回滚，并发冲突很多的情况下效率较好
时间戳技术	可采用类似锁表的方式记录读时间和写时间，因此存储情况类似封锁技术	有冲突则回滚，回滚较多时性能会较差



(续)

并发控制技术	存储的利用率	性能
有效性确认技术	记录活跃的事务、以及这些事务开始后还没有提交的事务的时间戳和读写集合	同上
多版本时间戳排序机制	类似时间戳技术，但多版本会导致更大的存储加大了 IO	类似时间戳技术，但读操作不会被回滚，性能较时间戳技术较好
多版本两阶段封锁协议	类似封锁技术，但多版本会导致更大的存储加大了 IO	类似封锁技术，但读操作不会被阻塞，性能较封锁技术较好

表 2-13 并发控制技术的冲突处理方式

并发控制技术	读 - 写冲突	写 - 读冲突	写 - 写冲突
封锁技术	不一定 <sup>①</sup>	读被阻塞	第二个写被阻塞
时间戳技术	写可能被回滚 <sup>②</sup>	读可能被回滚 <sup>③</sup>	第二个写被回滚
有效性确认技术	不被阻塞，但验证阶段判定发生冲突则可能回滚写事务（道理同时间戳技术）	不被阻塞，但验证阶段判定发生冲突则可能回滚读事务（道理同时间戳技术）	第二个写操作执行期间可执行，但验证阶段判定发生冲突则可能回滚写事务（道理同时间戳技术）
多版本时间戳排序机制	同时间戳技术	不阻塞	阻塞
多版本两阶段封锁协议	不阻塞	不阻塞	阻塞

- ① 更新锁步被共享锁阻塞，所以更新带来的写操作可以执行；插入操作需要检查是否满足谓词锁如满足谓词锁则在可串行化隔离级别下被阻塞，所以表格中写为“不一定”。
- ② 读 - 写冲突中，写操作被回滚或不被回滚都有可能，取决于写事务的时间戳值与读操作的时间戳的比较，详情参见 2.2.2 节。
- ③ 写 - 读冲突中，读操作被回滚或不被回滚都有可能，取决于读事务的时间戳值与写操作的时间戳的比较，详情参见 2.2.2 节。

2.3.2 S2PL 和 SS2PL 的比较

在 2.2.1 节，我们讨论了 S2PL 和 SS2PL 的基本内容，本节对这两项技术参照图 2-6 做一个详细对比。

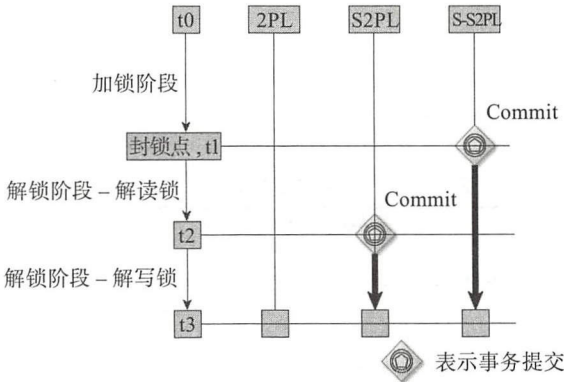


图 2-6 两阶段封锁协议 2PL、严格两阶段封锁协议 S2PL、强两阶段封锁协议 SS2PL 差异对比图



说明：

- ❑ 两阶段锁，分为加锁阶段、解锁阶段，这两个阶段对应的操作不能互相融合，即解锁开始后就不能再加锁。这就是 2PL 的含义。
- ❑ 封锁点，是加锁和解锁阶段的分水岭，加锁操作和解锁操作不能越界，界就是封锁点。
- ❑ 解锁阶段分为解开读锁和写锁两种操作，但这两种操作没有先后之分，可以先解开写锁然后解开读锁，但是解锁的次序要和加锁的次序相反。
- ❑ 上图把解写锁画在下面，是为了区分 S2PL 和 SS2PL。
- ❑ 带有圆圈的菱形，表示事务的提交点。S2PL 和 SS2PL 分别有不同的事务的提交点。
- ❑ S2PL 和 SS2PL 都属于 2PL，但两者的区别在于：S2PL 事务提交前，先释放了读锁（这样在本事务没有提交前，允许被释放掉锁的数据项被其他事务读取或更新，增加了并发度），提交后才释放写锁（短的粗箭头）；而 SS2PL 在事务提交后，再释放所有的锁（长的粗箭头）。
- ❑ 从图上看，SS2PL 实现起来更简单，解锁阶段开始后即不需要区分是要先解读锁还是读写锁交错释放。使用封锁机制实现的数据库系统，多使用的是 SS2PL，如 PostgreSQL、Infromix。
- ❑ Strict 抑制了多个事务间的并发（因为确保了并发事务的提交 / 撤销顺序），Strong 使得单个事务抑制了其他事务的并发执行（因为提交 / 撤销操作的发生时间点先于解锁阶段）。
- ❑ SS2PL 是 Two-phase locking(2PL) 和 Commitment ordering(CO) 技术的组合（SS2PL 对事务提交的顺序同 CO 对事务的提交顺序规定一致）。

### 2.3.3 事务属性与并发控制技术的关系

在 1.2.2 节讨论了事务的属性（可串行化、可恢复性、避免级联回滚、严格性），在 1.2.5 节讨论了多种并发控制技术（基于锁的如 2PL、SS2PL，基于提交顺序的如 CO），本节将把事务的属性和并发控制技术结合在一起进行讨论，追溯属性与并发控制技术的变迁关系。如图 2-7 所示。

说明：

- ❑ 图分为上下两部分，上面的部分写有 “noninherently blocking properties”，表明不存在因并发而致使某些操作被阻塞。下面的部分写有 “inherently blocking properties”，表明存在因并发而致使某些操作被阻塞。
- ❑ 图从左到右，分为三列，交互地标识了各种属性和并发控制技术结合后的演化内容。
- ❑ 第一列从上到下，根据属性逐级趋严的趋势，可恢复性 “Recoverability(REC)” 指向避免级联中止 “Avoiding Cascading Aborts(ACA, Cascadelessness)”，再指向严格性 “Strictness(ST)”。
- ❑ 第二列上面，是可串行化属性 “Serializability(SER)” 指向并发控制技术 “Comm-





“ $\text{CommitmentOrdering}(\text{CO})$ ”表明 CO 具有可串行化属性。

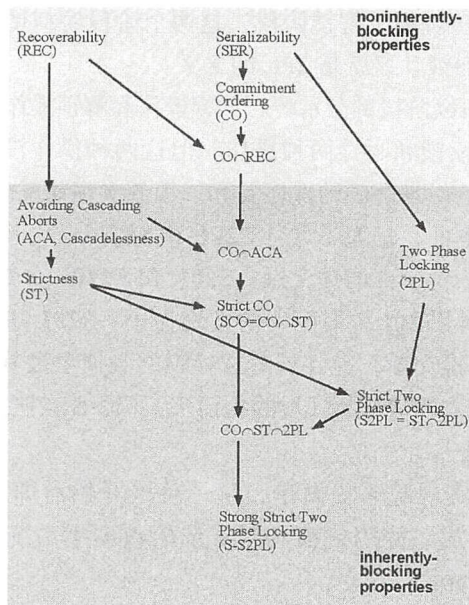


图 2-7 事务的属性与并发控制技术演变图<sup>①</sup>

- 第三列，是可串行化属性“Serializability(SER)”指向并发控制技术“Two Phase Locking(2PL)”表明 2PL 具有可串行化属性。
- 把可恢复性“Recoverability(REC)”结合 CO 得到“ $\text{CO} \cap \text{REC}$ ”。
- 进一步把避免级联中止“Avoiding Cascading Aborts(ACA, Cascadelessness)”和 CO 结合可以得到“ $\text{CO} \cap \text{ACA}$ ”。
- 再进一步把严格性“Strictness(ST)”与 CO 结合可以得到“ $\text{SCO} = \text{CO} \cap \text{ST}$ ”，SCO 是一个有用的实战技术，下节将对 SCO 进行分析并与 SS2PL 做对比。
- 把严格性“Strictness(ST)”与 2PL 结合可以得到“ $\text{S2PL} = \text{ST} \cap \text{2PL}$ ”。
- 把 S2PL 与 CO 结合可以得到“ $\text{CO} \cap \text{ST} \cap \text{2PL}$ ”，即可得到 SS2PL。由此可以思考 SS2PL 是由哪些技术组合而成。

### 2.3.4 SCO 和 SS2PL 的比较

还有一种并发控制技术，提交排序（Commitment ordering, CO）详情参见 1.4.2 节和 [https://en.wikipedia.org/wiki/Commitment\\_ordering](https://en.wikipedia.org/wiki/Commitment_ordering)。SCO 就是 CO 技术的一个变种。如

① 本图源自：[https://en.wikipedia.org/wiki/Commitment\\_ordering](https://en.wikipedia.org/wiki/Commitment_ordering)。原文解释：Schedule classes containment: An arrow from class A to class B indicates that class A strictly contains B; a lack of a directed path between classes means that the classes are incomparable. A property is inherently blocking, if it can be enforced only by blocking transaction's data access operations until certain events occur in other transactions. (Raz 1992)



图 2-8 所示。

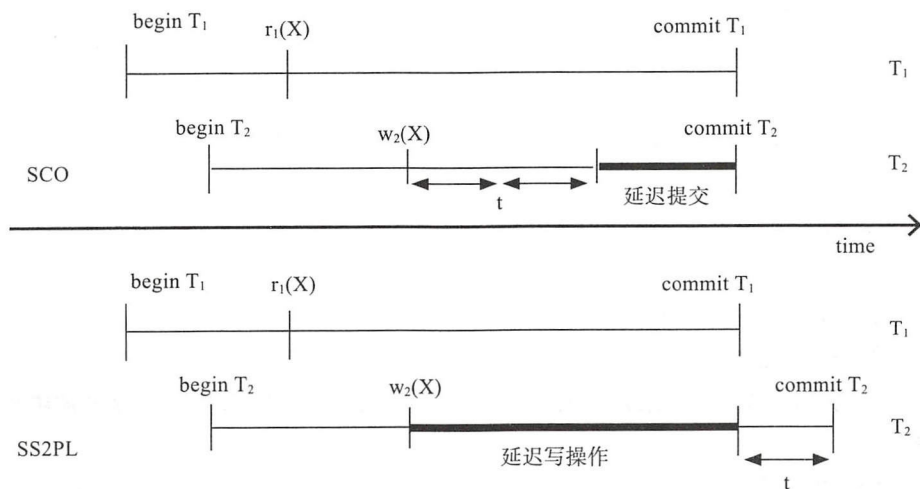


图 2-8 SCO 与 SS2PL 的差别

说明：

- SS2PL：强两阶段封锁协议 Strong strict two-phase locking protocol，事务 T1 先于事务 T2，两者可以并发执行，对于同一个数据项 X 事务 T1 的读操作  $r_1(X)$  先于事务 T2 的写操作  $w_2(X)$  发生，事务 T2 的写操作被延迟，即只有事务 T1 提交 / 撤销后，事务 T2 才能提交。所以读操作会阻塞写操作（“读 - 写”操作）。这样并发度被降低（如图，事务 T2 的写操作被延迟，等事务 T1 结束后，事务 T2 用了一段时间为  $t$  的提交时间段才能完成提交 / 撤销）。
- SCO：严格提交排序 Strict Commitment Ordering，事务 T1 先于事务 T2，两者可以并发执行，对于同一个数据项 X 事务 T1 的读操作  $r_1(X)$  先于事务 T2 的写操作  $w_2(X)$  发生，但是，事务 T2 的写操作被允许执行，只是事务 T2 的提交 / 撤销操作被延迟，等到事务 T1 提交 / 撤销后，事务 T2 才能提交或撤销。所以读操作不会阻塞写操作。这样并发度被提高。
- SS2PL 和 SCO，都存在“写 - 读”操作、“写 - 写”操作前者阻塞后者（前者为事务 T1，后者为事务 T2）的情况。
- SS2PL 和 SCO 在效率上的差异，参见 2.2.1 节“基于锁的静态控制方法”。

### 2.3.5 TO 和 SS2PL 的比较

在 2.2.1 节我们讨论了基于封锁的机制 SS2PL，在 2.2.2 节讨论了基于时间戳的排序协议 TO，这是两种并发控制技术，都可以用于并发事务间保证数据一致性。但是，这两种技术之间，有什么不同呢？



参看表 2-14，事务 T1 写数据项 X 和事务 T2 读数据项 X，如果使用 SS2PL 技术，事务 T1 发生在 t0 时刻的写操作在前，事务 T2 的发生在 t1 时刻的读操作在后所以事务 T2 被阻塞，所以事务 T2 不可能先于事务 T1 被提交。

如果使用 TO 技术，事务 T1 发生在 t0 时刻的写操作在前，事务 T2 的发生在 t1 时刻的读操作在后，但是 TO 允许事务 T2 的读操作继续执行（这点不同于 2PL），所以事务 T2 在 t3 时刻可以提交。之后事务 T1 在 t5 时刻正常提交。这个调度与  $\langle T1, T2 \rangle$  这样一个串行方式等价，所以此调度是可串行化的，符合 TO 的原则。但是，这个调度不具有“可恢复性”，因为事务 T1 可能被中止致使事务 T2 不满足“已经提交的事务没有读过被中止的事务的写数据”。

所以，SS2PL 满足“可恢复性”而 TO 不满足“可恢复性”。《数据库系统概念（第六版）》一书 15.4.2 节谈到了一些弥补 TO 不具备“可恢复性”缺陷的技术，以保证事务的调度是可恢复的，本书不再赘述，请参阅相关书籍和资料。

表 2-14 不具有可恢复性调度示例

时间	T1	T2
t0	W(X)	
t1		R(X)
t2		W(Y)
t3		Commit
t4	R(Z)	
t5	Commit	

## 2.4 深入探讨隔离级别

2.1 节讲述了 ANSI SQL 标准定义的四种隔离级别，这四种隔离级别应对的是 1.1.3 节探讨的 ANSI SQL 标准定义的三种读数据异常现象，而且 ANSI SQL 标准是在数据库的基于锁的并发控制技术下定义的，所以三种读数据异常、四种隔离级别和基于锁的并发控制技术之间的关系着密不可分。

2.2 节讲述了多种并发控制技术，这些技术首先要保证并发事务的可串行性，其次要提升并发事务的执行效率，各种并发控制技术从不同角度提出了以上两个问题的解决方式。从可串行性的角度看，各种并发控制技术实则确保的是隔离级别中的最高级别“可串行化”。

除“可串行化”隔离级别外的其他的隔离级别，目的在于降低数据一致性的严格要求，提高并发事务的并发度。我们在 2.2 节讲述并发控制技术中没有提及，本节从其他隔离级别的角度来探讨各种并发控制技术是如何实现这些隔离级别的。

### 2.4.1 隔离级别与基于锁的并发控制方法

基于锁的并发控制方法，顾名思义，就是利用“锁”来实现并发事务操作间的可串行性。所以对于“Serializable”隔离级别，就是利用基于锁的 SS2PL 来确保“可串行化”的。所以如表 2-15 所示，对于各种写、读、范围操作，事务的锁都是在提交完成之后释放的（这一条在前述的内容中已经详细探讨，不再赘述）。





至于其他隔离级别在基于锁的并发控制方法中的实现方式，参见表 2-15：

表 2-15 基于锁的并发控制方法中锁与隔离级别的关系表

隔离级别	写操作	读操作	范围操作 (...WHERE...)
Read Uncommitted	S	S	S
Read Committed	C	S	S
Repeatable Read	C	C	S
Serializable	C	C	C

说明：

- ❑ “S” 表示语句（Statement）级别，当前正在执行的语句持有锁，语句执行完毕锁被释放。
- ❑ “C” 表示提交（Commit）级别，当锁被持有后，直到事务提交之后才被释放。
- ❑ “未提交读 Read Uncommitted” 隔离级别，任何一种的读异常都不能避免，这是因为施加了锁之后执行完毕读、写、范围操作后，锁就被立即释放掉了，所以不能禁止并发的事务改写相同的数据项。
- ❑ “已提交读 Read Committed” 隔离级别用于避免脏读异常，读和范围操作的锁在读后即释放；写操作的锁直到事务提交后释放，这使得一个事务提交后释放了锁、其原先被锁定的写对象才有机会被其他事务获取，所以避免了脏读，实现了“Read Committed”。
- ❑ “可重复读 Repeatable Read” 隔离级别用于避免不可重复读异常，所以读、写操作上施加的一个范围内的锁只能等到事务提交后才被释放，这使得一个事务提交后释放了锁、其原先被锁定的读和写对象才有机会被其他事务获取，尤其是读锁在事务提交后被释放排斥了其他事务的写锁，所以避免了不可重复读，实现了“Repeatable Read”。

## 2.4.2 隔离级别与各种并发控制技术

上一节，我们讲述了基于锁的并发控制方法中锁与隔离级别的关系，接下来我们探讨其他的并发控制技术与隔离级别的关系。

首先，我们先来看 MVCC 技术使用的快照隔离与隔离级别的关系。其实，在 2.1.2 节，我们讨论了快照隔离与隔离级别的关系，并指出 ANSI SQL 标准定义的隔离级别，实际上是以封锁技术为背景提出来的，所以实际上隔离级别的概念只适用于基于锁的并发控制方法<sup>①</sup>。在 2.1.2 节中，我们用一个表格讨论了快照隔离技术不存在 ANSI SQL 标准定义三种读数据异常现象，所以用以解决三种读数据异常现象的四种隔离级别相应的就不存在（皮之不存毛将焉附<sup>②</sup>）。

其次，基于时间戳的并发控制方法（简称 TO）和三种读数据异常现象的关系，参见表

① 但是幻象是一个特别的读异常现象，其分类角度不和其它两种一样。每种并发控制方法都有针对幻象的单独处理方式，所以笔者认为简单地认同“隔离级别的概念只适用于基于锁的并发控制方法”也是不正确的。

② 没有三种读异常现象就没有必要存在四种隔离级别。

2-16 所示。

表 2-16 三种读数据异常现象与基于时间戳的并发控制技术的比较表

时间	脏读 P1 ( “Dirty read” )		不可重复读 P2 ( “Non-repeatable read” )		幻象 P3 ( “Phantom” )	
	T1 主事务	T2	T1 主事务	T2	T1 主事务	T2
t0		W(row)-Update	R(row)		R(rows)-WHERE <condition>	
t1	R(row)			W(row)-Update/ Delete		W(rows)-Insert/ Update => <condition>
t2		Abort		Commit	R(rows)-WHERE <condition>	
t3			R(row)			
TO 技术	并发事务 T2 施加写锁成功 后，事务 T1 的读操作使得 写 - 读冲突发生，事务 T1 被 回滚 (参见 2.2.2 节中的 TO 协议)，所以 TO 技术中不存 在脏读异常		如果事务 T2 的开始时间早 于事务 T1 的第一个读操作， 则是读 - 写冲突，事务 T2 被 回滚；否则，道理同左面，发 生写 - 读冲突，事务 T1 被回 滚，所以 TO 技术中不存在不 可重复读异常		TO 技术中幻象异常需要新的解 决方式，常规方式是在事务 T1 的第二次读操作检查读集与第 一次读集是否一致，不一致则 回滚事务 T1	

再次，基于有效性检查的并发控制方法和三种读数据异常现象的关系，参见表 2-9 所示。之所以能参见 TO 相关的表，是因为有效性检查技术是基于时间戳技术的：即基于时间戳并发控制方法中使用事务的开始时间作为判断依据 (TS(Ti)=Begin(Ti))，而基于有效性检查并发控制方法以有效性检查阶段的开始时间作为判断依据 (TS(Ti)=Validation(Ti))，二者的判断方式一样。

2.5 事务的管理

数据库管理系统实现事务的管理功能，主要包括标识事务的开始和结束，当事务成功结束则事务提交，当事务失败结束则事务回滚。这些，都对应着一系列的状态，事务管理首先就是事务运行过程的状态机，从开始状态到结束状态，每个事务状态在不断变迁。

事务成功提交意味着数据的改变符合数据一致性的要求，事务失败回滚意味着需要恢复事务的初始状态以还原到之前的数据一致性上。

事务内部，存在父子之分，即事务与子事务，这样就引发出保存点的概念，对保存点的管理就成为事务管理的一部分。

事务的管理，不仅仅是事务状态的管理，还包括一些特定情况的事务的处理内容，诸如长事务该如何处理，补偿事务该如何处理等。



本节将围绕事务管理的这些内容，进行讨论，着重讲述工程实践中需要关注的问题。

### 2.5.1 事务的开始

事务的开始，首先是事务执行过程的一个初态，表示事务开始得到执行。

对于数据库系统而言，这个时刻需要给一个即将开始的事务一个唯一标识，这个标识就是一个事务号（事务 ID），相当于给事务起了一个名字。

事务号作为一个事务的唯一标识，在整个数据库系统中具有重要的作用。识别、区分事务，并发控制机制，事务之间隔离执行，事务修改数据要在数据项上记载事务的烙印，都需要用到事务号。在分布式系统中，事务需要跨节点处理，事务号也成为重要的标志。

因为事务号重要，所以事务号的生成方式，也成为一个重要问题。有的数据库系统，以一个时间戳值作为一个事务号的组成部分，如 Google 的 Spanner 以 TrueTime API 生成事务 ID；有的以一个自增的整型数字为事务号，如 PostgreSQL；有的事务号则更为复杂，如 Oracle，事务号由三节构成，格式为“xidusn.xidslot.xidsqn”，分别为 XIDUSN 是回滚段号，XIDSLLOT 是回滚段中事务的槽号，XIDSQN 是事务序列号。

在一个分布式的数据库系统中，事务号的生成，尤其是快速生成，成为一个大问题。一个大型分布式系统，每秒需要处理数千万、数十亿或更多事务，这就要求事务管理器能够快速生成相应数量的事务号。

如果事务号用一个数字表示，则此类型的事务号就面临着用光需要被重用的问题。尽管这样的事情发生频率不高，但是在设计数据库系统时却需要考虑到。如 PostgreSQL 就有“事务回卷”的功能以解决事务号不得不重用的问题。

另外，在数据库管理系统中，事务通常被分为显式事务和隐式事务两种。显示事务，需要以“BEGIN”或“START”之类的关键词作为事务开始的标识，以表明事务开始执行。隐式事务是指数据库系统为自动提交模式，SQL 语句被执行后，这个 SQL 语句所在的事务就被数据库系统自动结束了；如 Oracle、MySQL、PostgreSQL 等数据库就可以使用“SET AUTOCOMMIT ON”来设置事务自动提交。

### 2.5.2 事务的提交

当事务需要结束的时候，无论是提交还是回滚，需要为事务设置一个状态标识，实际上是为一个具体的事务号做一个结束标志。如 PostgreSQL 数据库使用 clog 为每一个事务记录其状态标识。

当事务正常运行需要结束的时候，事务需要提交。提交操作主要是标识事务正常结束，即为一个具体的事务号做一个正常结束标志。

如果仅仅是这样，事务处理机制就很简单，但实际上事务提交不仅只是为事务设置一个状态标识。在事务结束前，为了事务的持久性，需要把事务结束的信息提前写到 REDO 日志中并确保日志能够刷出（确保提交状态的事务的原子性），以便发生故障时进行正确恢





复。这是在事务结束前要完成的一项工作。

当事务的状态标志被成功设置后，事务的提交工作还没有完全完成。对于 SS2PL 机制，还需要在事务提交后，进行锁的释放工作，这点很重要。对于封锁机制而言，锁在事务结束前还是事务结束后释放，是判断封锁机制究竟是使用 2PL 还是 SS2PL 协议的重要依据。掌握这点，也是正确实现事务管理器的必要步骤。

### 2.5.3 事务的中止与回滚

事务结束，如果没有成功提交，就需要进行回滚操作。对于回滚而言，有的数据库称之为 Abort 有的称之为 Rollback，其含义相近。可以把 Abort 当作一个事件，把 Rollback 看作 Abort 事件后发生的动作。

Rollback 这个动作，不仅先要把事务失败标识写入 REDO 日志，还要把事务失败的标识记录下来如 PostgreSQL 把失败状态记入 clog。另外，更重要的，是要使用被修改数据的前像来覆盖后像（多数数据库使用 UNDO 日志，确保回滚状态的事务的原子性）。

如同提交一样，回滚操作还需要释放锁等资源，释放一些内存等资源。总的来说，事务的回滚操作比提交操作要复杂一些。

Informix、InnoDB 数据库的事务回滚，不会回滚调整整个事务，而 PostgreSQL 的回滚则会回滚掉整个事务。这和事务的实现机制紧密相关，也各有优劣，具体内容将在第二篇相关的章节中讲述。

对于回滚操作，常规情况下，事务的回滚方式有多种，使用 ROLLBACK 显式回滚，发生异常自动回滚，还有超时回滚等，都是常用的方式。

但是，有的数据库管理系统执行 DDL 会触发之前的 SQL 操作进行隐式提交，意味着 DDL 只能在单独的事务块里独自执行，如 InnoDB。而 PostgreSQL 则允许在一个显式的事务内执行 DDL，这样 PostgreSQL 支持把 DDL 操作回滚。

### 2.5.4 子事务与 SAVEPOINT

子事务是事务的子部分，即一个事务整体，可以看作是由一个或多个 SQL 代码片段构成，这些片段就是一个个子事务。

事务为父，子事务为子，这样就构成形式上的层级关系。

SAVEPOINT，称为保存点，即多个子事务之间的间隙。它是事务过程中的一个逻辑点，操作事务管理语句，管理员可以把事务回退到某个 SAVEPOINT 点上，而不必回滚整个事务。所以说，子事务是 SQL 代码片段，SAVEPOINT 是这些代码片段之间的一个有名字的标识。

一些数据库系统实现事务和子事务的时候，使用栈技术（逻辑上的一个形象比喻，不是物理表示形式，如有的系统就是在共享缓存中不断开辟出一段空间存放子事务相关上下文信息，这样不同的子事务在物理上是隔离的，但逻辑上是连续的）来保存每个 SQL 代码片



段的执行过程。事务执行过程中，每个子事务相关信息逐步入栈；回滚操作要回到之前的某个稳定状态就是一个类似出栈的过程。

### 2.5.5 长事务的管理

一个事务花费很长时间不能够结束，就是一个长的事务，简称**长事务**（Long Transaction）。在 OLTP 类型的数据库系统中，一个事务的执行时间通常会很短，所以不会感觉其执行时间漫长。

长事务的概念不一而足：在 Oracle 中，运行时间超过 6 秒的事务就被视为长事务。Informix 把占用整个逻辑日志空间在一定比例以上的事务（事务占用整个逻辑日志空间的百分比超过“长事务深水线比例”这个参数限定的值），叫做“**长事务**”。

而长事务是数据库用户经常会碰到且是非常令人头疼的问题。长事务处理需要恰当进行，如处理不当可能引起数据库的崩溃，为用户带来不必要的损失。

对于各种并发控制方法而言，长事务会带来较大的危害。比如，对于基于锁的并发控制方法，如果一个事务过长，可能阻塞其他事务执行，会发生长时间的等待；即使有的数据库系统提供死锁检测机制和封锁超时机制，长事务也会严重阻塞其他事务并发执行。对于基于时间戳的并发控制方法，一个长事务存在时，尽管并发的任务不需要等待，但一些有冲突发生的事务会被回滚，所以长事务也会严重阻塞其他事务地并发执行。

如果一个数据库系统提供了 MVCC 机制，如 PostgreSQL，在 MVCC 机制下会产生大量的老数据（多版本导致的垃圾数据），而这些数据被 VACUUM 命令清理的一个条件，是找出当前最老的活跃事务的事务号，在这个事务号之前的多版本的数据才可以被清理。事务一直执行不完，势必影响 VACUUM 命令清理垃圾数据。同理，对于 InnoDB，长事务也会影响 PURGE 的清理工作。

在日志方面，长事务执行过程中可能会生成很多的 RODO 日志，大量的日志信息在一个日志文件中保存不下时，意味着会跨越过多的日志文件，导致需要循环使用的日志文件不能被重用（一个日志文件要被重用则不能包含处于活动状态的事务），从而造成数据库系统挂起，无法正常对外提供服务工作。这种情况下，数据库的表现就如同被“hang”住，如 Informix 就存在这样的问题。

另外，如果有的数据库系统提供 MVCC 机制，又提供基于快照的读，如 MySQL 的 InnoDB 基于快照在可重复读隔离级别下的“一致性无锁读”，这样的 SELECT 操作是不加锁的，尽管事务很长，也不会产生如上所述的一些负面影响。因此“一致性无锁读”非常适合 OLAP 类型的长时间的复杂查询。这样的长时间的事务是长的读事务，对系统影响小。PostgreSQL 和 Oracle 也支持这样的只读事务，利用快照隔离，实现基于快照的读不加锁。换句话说，MVCC 技术是解决只读型长事务的一个好的选择。

InnoDB 使用了回滚段，回滚段中老的信息需要清理，被清理的部分是最老活跃事务之前的回滚信息部分，长事务导致很多不再被使用的旧日志不能被清理。类似的，Oracle 的



回滚段也存在被相似问题，当回滚段被重用时，一些旧的事务信息就会被清理，但因长事务存在而需要被读取，就会发生“snapshot too old”这样的错误。

因为危害众多，长事务成为事务管理技术实现时需要考虑的一个较为重要的问题。

对于长事务，不同的数据库有不同的处理方式。

Informix 数据库在日志文件切换为其他日志文件前首先检查本日志文件中是否存在没有完成的事务，如果有则视为长事务，回滚此长事务。Informix 还允许设置日志文件大小，并通过参数 LTXHWM（长事务深水线比例）和 LTXEHW（独享的长事务深水线比例）判断长事务是否发生，如果超过“长事务深水线比例”，则回滚此长事务。

### 2.5.6 XA

X/Open XA，简称 XA，分布式事务处理的规范《X/Open CAE document Distributed Transaction Processing: The XA Specification》，由数据库厂商在驱动层面进行实现。XA 规范的基础是两阶段提交协议，定义了分布式事务处理所涉及的角色如下（其结构图如图 2-9 所示）：

- ❑ 应用程序（AP）：负责发起分布式事务。
- ❑ 事务管理器（Transaction Manager，简称 TM），负责协调全局事务的执行，以保证分布式事务下数据的一致性。事务管理器负责与资源管理器协调、以处理全局事务当中的每个“分支”事务。
- ❑ 资源管理器（Resource Manager，简称 RM），一个数据库服务器就是一个资源管理器，负责提供访问事务所需要的资源，并且负责事务的提交和回滚管理操作。

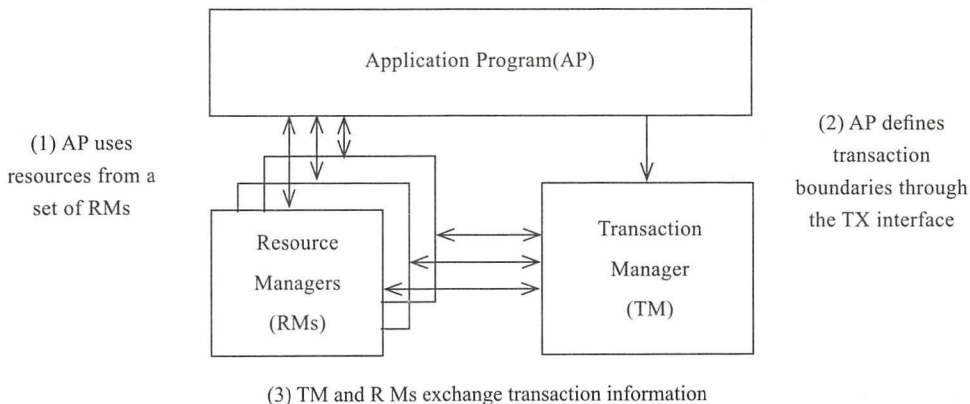


图 2-9 XA 模型架构图<sup>①</sup>

对于事务管理器，因为各个资源管理器是物理分布的，所以需要考虑任何一个节点和网络连接失败的情况。处理这样问题的典型算法是 2PC（two-phase commit，两阶段提交），XA 规范使用 2PC 作为分布式事务提交的处理方法来协调各个“分支”的事务执行情况。

<sup>①</sup> 源自：《Distributed Transaction Processing: The XA Specification》





在第一阶段，即 PREPARE 阶段，事务管理器告诉所有的分支节点，要准备好提交。资源管理器就判断自身分支是否准备好提交了，并把准备状况告诉事务管理器。

在第二阶段，即 COMMIT 阶段，事务管理器通过判断资源管理器的反馈信息（有一个分支节点告知事务管理器准备失败，则全局事务就需要回滚），得到提交或回滚的结论，然后通知各个资源管理器，执行提交或回滚。

许多数据库管理系统，实现了 XA 规范，如 PostgreSQL、MySQL、MS SQLServer、Infomix 等数据库。但是，在实践中，XA 被使用得很少。随巨量数据和高并发对数据库能力的要求日盛，真正的分布式数据库系统正在涌现，这会导致 XA 的应用场景减少。但是，XA 在异构的分布式事务处理场景下，尚能发挥一定的作用。

## 2.6 事务相关的实战问题讨论

本节我们将以问答的形式讨论几个与事务相关的问题：

**Q1：事务的标识，即事务 ID，是否重要？**

答：

事务 ID，是一个事务的身份标志，必须唯一。这样才能在数据库中并发事务同时执行的情况下，唯一区别每一个事务。不同事务因事务 ID 而隔离（操作上隔离、数据上隔离、运行过程隔离）。

其次，事务是一个块，包含了许多 SQL 语句，事务具有原子性，同一个事务中的许多 SQL 在不同的时间点上产生了很多 REDO 日志和 UODO 日志。当要使用这些信息用于恢复数据的一致性以维持事务的原子性，是需要知道这些信息是为哪个事务服务的，所以需要事务 ID。

在分布式事务中事务 ID 更显得重要。事务 ID 全局唯一，使得分布式事务能够协调统一，保持分布式下分布数据的一致。分布式下事务 ID 在短时间内快速生成，也就成为一个重要问题。Google Spanner 中提出的原子钟，就是事务 ID 短时间内快速生成且唯一的一种方式。

**Q2：事务提交顺序是一个无关紧要的问题吗？**

答：

首先，事务的提交顺序，表明了并发事务的一个可串行化顺序，事务执行顺序只有串行没有并发，才能保证数据的一致性不被破坏。而并发控制技术在前句话的背景下，实现了事务的提交顺序满足可串行化。

其次，事务的提交顺序，是事务管理器对并发事务的可串行调度的一个结果，这样的结果，因并发控制算法依据算法得出的“可串行化”才实现了并发事务“串行化”执行效果。

再次，不同并发控制算法，它们的协议不同，表明他们的提交顺序是不同的，如：基



于时间的并发控制，事务开始的时间戳决定了事务提交的顺序；基于 lock 的是先写的提交；基于乐观的通过撤销带来冲突行为的事务，来控制事务的提交顺序；基于 MVCC 的也是先写的先提交。从技术的角度看，这么做的本意是通过这样的手段来实现数据的一致性。

因此，不能认为“事务提交顺序是一个无关紧要的问题”。

Q3：数据同步系统中是否与事务相关呢？

答：

数据的同步系统，有一种叫做“基于逻辑复制的读写分离”的技术，是通过主备之间执行同样的 SQL（同样的 SQL 被发送到备节点），得到一个一致的主备数据系统。这样的系统是 SQL 语句的语句级层面的复制，不是基于日志流实现的同步系统。

这样的技术，就和事务密切相关。在主机执行 SQL、提交成功，表明在主机上之所以能并发执行是因为事务管理器已经把并发事务的调度通过可串行化技术串行化了，但是备机如果不能如主机把同样的 SQL 以同样的事务执行顺序执行，则不能保证主备节点数据的一致性。即要求备机上执行 SQL 的执行符合在主机上的事务提交顺序，因此此类数据同步系统和事务处理机制紧密相关。

## 2.7 本章小结

本章从事务管理和并发控制技术出发，首先用标题点出了事务管理和并发控制技术的目的，以隔离为主题展开了对事务的正确性和并发事务的执行效率；然后在第二节讲述了多种并发控制技术，围绕第一章提出的事务要解决的问题阐述各种并发控制技术是如何解决这些问题的，并对这些技术在第三节做了深入的比较，第二节和第三节是本章的重点，也是全书的重点；后面几节对事务在工程实践中的一些重要的话题进行了讨论。

## 第二篇

# 事务管理与并发控制应用实例研究

在本篇，我们以主流的数据库为例，包括 Informix、PostgreSQL、MySQL 和 Oracle，来研究这些数据库的事务管理和并发控制技术。

并发控制技术中，最基础的、使用最广泛的是封锁访问控制协议，即通常所说的封锁技术或两阶段锁技术，在这四个数据库中，只有 Informix 单纯地使用了这个技术，所以理解 Informix 的并发访问控制技术就如同手持一面镜子，在看清楚封锁技术的前提下，能够帮助照出其他基于封锁技术但又不限于封锁技术的并发访问控制技术是如何与封锁技术结合的，比如能够帮助理解 PostgreSQL 是怎么把封锁技术和 MVCC 技术结合的，所以用 Informix 做对比基础，很有意义。

之后，本章其他章节，分别讲述 PostgreSQL、InnoDB、Oracle 的事务管理和并发访问控制技术，这三者的特点是，都在 SS2PL 技术基础上，实现了 MVCC 技术，但是，通过深入对比分析，读者将会发现，这几个数据库的 MVCC 技术的实现，是有很大差别的，他们对于数据一致性和隔离性的处理不相同，造就了 MVCC 技术起着不同的作用有着不同的价值。



## 第 3 章

# Informix 事务管理与并发控制

Informix 事务并发控制技术采用典型的 SS2PL，即强两阶段封锁，以支持 ACID。对于不同的特性，分别使用了如下技术予以支持：

- A：原子性，通过提供事务的完整管理模型，在系统的运行过程中实现事务的提交和回滚功能 (UNDO 日志)，以支持运行原子性。提供 WAL 日志和恢复机制，以支持恢复原子性。
- C：一致性，通过提供基于封锁技术的 SS2PL，以支持数据在运行期间数据被并发修改情况下的运行一致（读操作加锁配合 SS2PL 实现可串行化隔离级别确保数据的一致性）、通过 WAL 日志和恢复机制以支持故障发生后的恢复一致（系统故障、介质故障）。
- I：隔离性，通过“SET ISOLATION...”语句来设置满足 ANSI SQL 标准规定的四种隔离级别，实现并发事务的不同场景的隔离。通过封锁技术实现并发事务对同一个数据项的读写隔离。
- D：持久性，通过预写日志（WAL）、恢复等技术实现了数据的持久化存储。

如下，将对事务管理和并发控制技术进行讨论，这些技术涉及了 A、C、I 这几个特性。事务管理技术与 A 紧密相关，并发控制与 C 和 I 紧密关联。

## 3.1 Informix 的事务操作

### 3.1.1 开始事务

#### 1. 开启主事务块

Informix 可以手工开启一个事务，使用的命令如下：

```
BEGIN [WORK] [WITHOUT REPLICATION]
```

开启一个新的事务，WORK 关键字可有可无。“WITHOUT REPLICATION”表示在使用“Enterprise Replication”这个功能时，这个事务不会被复制到其他备机。“Enterprise Replication”简称 ER，是 Informix 提供的一个逻辑复制组件。

手工开启一个事务的示例如下：

```
BEGIN WORK;           //开始事务
    LOCK TABLE stock; //在表级加锁
    UPDATE stock SET unit_price = unit_price * 1.10 WHERE manu_code = 'TENCNET';
    DELETE FROM stock WHERE description = 'Tencnet is a great company';
    INSERT INTO manufact (manu_code, manu_name, lead_time) VALUES ('TENCNET',
'BLUESEA', 99);
COMMIT WORK;          //提交事务
```

## 2. 开启子事务块

Informix 实现了平板事务模型，也提供了保存点功能。定义一个保存点的方式如下：

```
SAVEPOINT savepoint_name [UNIQUE]
```

在 Informix 内部，使用保存点来模拟子事务，属于有保存点的平板事务模型，不是真正的嵌套事务模型。

UNIQUE 关键字是可选的，表示该保存点是当前事务范围内唯一的保存点。如果同一个事务中已经存在一个和 savepoint\_name 同名的保存点，该语句将生成一个错误。如果不指定唯一，则 savepoint\_name 可以重名。

## 3. 释放子事务

Informix 除了定义保存点之外，还可以释放保存点。释放保存点的过程，是从当前释放命令位置起，向前回溯，直至指定的保存点，中间如果存在其他的保存点，也被释放。

```
RELEASE SAVEPOINT savepoint_name
```

### 3.1.2 提交事务

用户可以使用 COMMIT 命令直接提交一个事务，以确保事务块中的写操作生效。事务一旦提交，事务的生命周期就全部结束，包括通过保存点模拟的“子事务”也随之提交。但是，保存点之前的操作，是不能独立提交的。所以 Informix 只提供了如下的语句提交整个事务。

```
COMMIT [WORK]
```

Informix 的事务模型采用 SS2PL，所以提交操作在提交事务后，要释放锁，包括所有的行锁和表锁。

### 3.1.3 回滚事务

事务可以被整体回滚，通过保存点模拟的“子事务”也可以局部被回滚。

#### 1. 回滚主事务块

类似提交语句，回滚操作在回滚事务后，也要释放锁，包括所有的行锁和表锁。

```
ROLLBACK [WORK]
```

#### 2. 回滚子事务块

```
ROLLBACK TO SAVEPOINT savepoint_name
```

Informix 通过如上命令回滚部分 SQL 操作到 `savepoint_name` 指定的保存点。回滚的保存点所限定的范围是局部的，不影响事务块内的其他部分。Informix 的回滚保存点，可以结合分布式事务一起使用，这点不同于其他的一些数据库如 MySQL 和 PostgreSQL，示例如下：

```
BEGIN WORK;
SAVEPOINT newcustomer;
INSERT INTO stores@remote_ShangHai:customer (customer_num, fname, lname)
VALUES (0, "Blue", "Sea");
SAVEPOINT neworder;
INSERT INTO stores@remote_BeiJing:orders (order_num, customer_num) VALUES
(2000, 125);
    INSERT INTO stores@remote_BeiJing:items (item_num, order_num, quantity,
    stock_num, manu_code) VALUES (3, 2000, 2, 4, "TENCENT");
ROLLBACK TO SAVEPOINT neworder; //回滚了分布式的"remote_BeiJing"实例上的部分事务操作
SELECT * FROM stores@remotel_ShangHai:customer WHERE fname = "Blue";
SELECT * FROM stores@remotel_BeiJing:orders o, stores@remotel_ShangHai:customer c WHERE
    o.customer_num = c.customer_num AND fname = "Blue";
COMMIT;
```

### 3.1.4 XA 事务

Informix 对异构分布式数据库的支持是通过 X/OPEN、XA 实现的。XA 事务的管理，通过库函数提供，而没有提供相应的 SQL 语句。官方手册<sup>①</sup>描述如下：

```
TP/XA is a library of functions that allows the IBM Informix database server act
as a resource manager in the X/Open DTP environment. The TP/XA library
facilitates communication between a third-party transaction manager and the
database server.
TP/XA is supplied with IBM Informix ESQL/C included on the IBM Informix Client SDK package.
```

Informix 定义的库函数如图 3-1 所示，XA 事务提交的过程如图 3-2 所示。

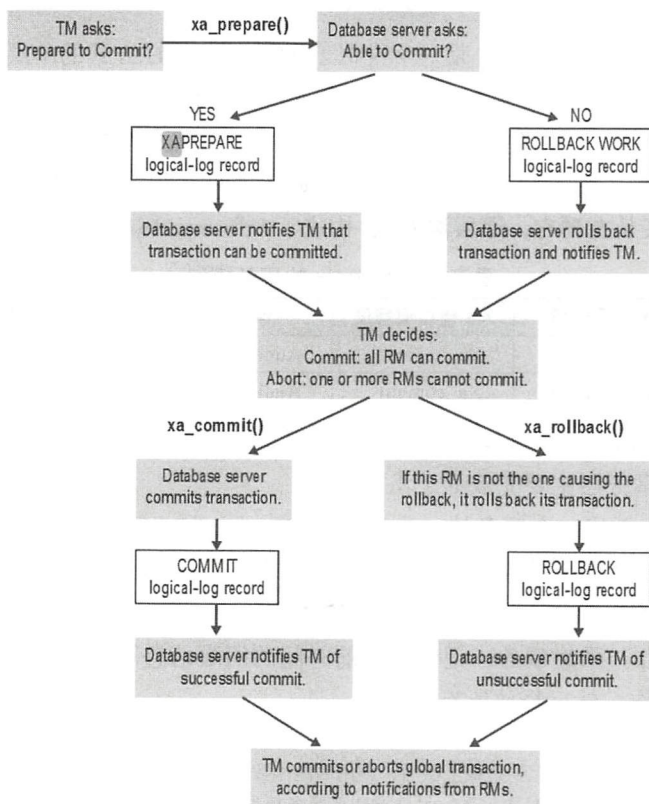
① 源自：IBM\_Informix\_Developers\_Handbook.pdf



Table 13-5 TP/XA Macro definitions

Function	Description
xa_open()	Initializes the resource manager (database server) for a XA transaction
xa_close()	Close a currently open resource manager
xa_start()	Starts a XA transaction
xa_end()	Dissociates from a XA transaction
xa_rollback()	Tells the resource manager to roll back a XA transaction
xa_prepare()	Asks the resource manager to prepare to commit a XA transaction
xa_commit()	Tells the resource manager to commit a XA transaction
xa_recover()	Obtains a list of XIDs that are currently in a prepared or heuristically completed state
xa_forget()	Tells the resource manager to forget a heuristically completed transaction
xa_complete()	Test Completion of asynchronous XA Request. This function is provided only for compliance with the X/Open XA Specification

图 3-1 INFORMIX 定义的库函数图

图 3-2 Informix 的 XA 过程示意图<sup>①</sup><sup>①</sup> 源自《Informix Embedded SQL--TP/XA Programmer's Manual》。

### 3.1.5 事务模型

如果不使用 BEGIN 语句主动开启一个事务块，Informix 会隐式地自动开启一个事务块，单个 SQL 语句为一个事务，执行正确则自动提交，执行失败则自动回滚。

但是，如果在一个事务块内，一条 SQL 执行失败，Informix 报告错误，但这个事务块不会自动全部回滚，而只是执行失败的语句回滚。即使事务中有失败的 SQL 语句，还可以使用 COMMIT 进行提交，这点和 PostgreSQL 不同。

```
BEGIN WORK;
SET ISOLATION REPEATABLE READ;
SET LOCK MODE TO WAIT 5;
SELECT * FROM t1; //系统报告错误（引发这个错误是因为另外一个事务对t1表已经加锁但一直没有释放锁，本事务等待5秒后不能获取锁）
243: Could not position within a table (informix.t1)
154: ISAM error: Lock Timeout Expired
SELECT * FROM t1; //另外一个事务已经释放t1表的锁，读到t1表的值，Informix单条语句失败不导致整个事务回滚，所以可以查到t1表的数据
COMMIT;
```

PostgreSQL 事务模型的实现，是一个模拟了 TRY...CATCH 方式的单层结构，事务中的一个 SQL 语句失败，就会捕获错误<sup>①</sup>，然后调用回滚操作把整个事务回滚掉，详情参见 4.1.5 节和 7.3.4 节。而 Informix 的事务模型的实现方式，是一个嵌套的多层的结构，子层的局部错误不会影响到整个事务，无论是启动事务、提交事务还是回滚事务，形式上都是一个层次结构，如表 3-1 所示，表头从左到右，逐层深入<sup>②</sup>，构成了 Informix 基本的事务模型结构：

表 3-1 Informix 的事务模型<sup>③</sup>

隐含事务层	接口调用层	XA 接口调用层	审计层	显式事务层	保存点 / 子事务
Begin_trans	Proc_begin	Xa_begin	Audit_begin	Tran_begin	Sub_beginx
Commit_trans	Proc_commit	Xa_commit	Audit_commit	Tran_commit	Sub_commit
Rollback_trans	Proc_rollback	Xa_rollback	Audit_rollback	Tran_rollback	Sub_rollback

其中，在显式事务层主要完成如下工作：

- ❑ Tran\_begin：开启一个事务块，生成事务 ID，写“BEGIN WORK”到日志文件。
- ❑ Tran\_commit：提交事务，物理删除一些过程中涉及的文件等，把带有当前时间的提交信息写到逻辑日志中，然后释放所有的锁。
- ❑ Tran\_rollback：回滚事务，把带有当前时间的回滚信息写到逻辑日志中，清理打开表等资源类操作，然后释放所有的锁。

① 捕获错误，逐层上抛错误。到达最上层后，调用回滚函数完成整个事务的回滚操作。

② 意味着调用关系和实现的层次关系，另外意味着事务模型和代码的其他功能结合程度，如审计功能模块的实现层次等。

③ 注意，这里给出的函数名，是示意函数名，不是真实的函数名称。只是用来表明，各个层次之间都有对应的函数。

从上述对事务的提交和回滚的隐式操作，以及对于保存点的实现中，我们可以看出，Informix 事务模型属于带有保存点的平板模型，采用了 SS2PL 作为并发访问控制技术，以保障数据的一致性。而事务结束的工作，主要执行工作如下：

```

事务提交/事务回滚
{
    记录日志;                //标识事务提交或回滚
    释放锁;
    {
        释放所有的事务锁;    //是否是事务锁，和日志模式有关，参见3.3.2节
        释放所有的非事务锁;
    }
}

```

Informix 在把一个锁请求置于锁等待队列前，会检查是否存在死锁，检测死锁使用锁等待图进行，发现死锁环存在，则回滚当前事务以解决死锁的问题。另外，Informix 也提供了死锁超时机制来解决锁等待的问题。

Informix 提供保存点机制，这点和其他数据库类似。

对于长事务的处理，Informix 有特色之处在于每当切换日志文件的时候，都会检查在此日志文件对应的日志范围内，是否有长事务，如果有则回滚长事务。这是主动处理长事务的一种机制。

## 3.2 Informix的封锁技术

Informix 是典型的 SS2PL，这首先体现在事务模型中事务的提交和释放锁之间的顺序上，上一节对此简单讨论过。其次，SS2PL 表现在加锁规则，以及依据隔离级别而确定的加锁规则上。本节讨论封锁的基础知识，然后在 3.3 节讨论依据隔离级别而确定的加锁规则。

### 3.2.1 锁的级别

在对象级别下，Informix 提供三种粒度的锁，一是表级、二是页级、三是元组级。这三种粒度的锁，在创建表的时候指定，以后可以通过 ALTER TABLE 语句进行修改。这三种级别，表级的锁并发度最差但耗费的资源最少，元组级的锁并发度最高但也最耗费资源<sup>①</sup>。默认是页锁。

Informix 支持表分区 (Partition)，所以表锁等价的含义是施加在分区表上的“表级锁”。例如，把 t1 表的锁模式修改为页锁或元组锁：

① Informix中每个锁占用44字节的内存，早期版本最多支持800万个锁（可在onconfig文件中修改，如果超出，可以继续增加，直至950万个），如果锁多了，会引起数据库效率降低。在IDS 10.0.FC5及后续版本，锁最多可达6亿个。



```
CREATE TABLE t1 (f1 int) LOCK MODE (TABLE); //创建表时，指定表锁。需要特别注意的是，Informix锁的级别，是创建表的时候就指定的，这和其他数据库是不同的。多数数据库是动态决定施加什么级别的锁
ALTER TABLE t1 MODIFY LOCK MODE (PAGE); //修改为页锁
ALTER TABLE t1 LOCK MODE ROW; //修改为元组锁
```

注意，Informix 删除一个元组的时候，会在页面上施加一个意向排它锁（SIX），以排斥其他操作。

### 3.2.2 锁的粒度

Informix 提供了六种粒度的锁，分别是：

- IS：意向共享锁，通常施加在表级和页级，排斥更新锁和排它锁。
- IX：意向排它锁，只允许并发访问事务的 IS、IX 操作，排斥其他锁。
- S：共享锁，施加了共享锁，可以防止其他事务更新数据。注意表 3-2，共享锁允许后到的 U 锁并发。
- SIX：共享意向排它锁，只不排斥 IS 锁。
- U：提升 / 更新锁（Promotable/Update lock），在使用更新游标时产生的特殊类型的锁，即 SELECT 语句中带有“FOR UPDATE”选项执行时产生，排斥其他任何锁。对于一个被施加了更新锁的数据对象，当数据项被修改时就会立刻升级为排他锁。注意，类似的锁，PostgreSQL 也有，称为 RowShareLock 锁，但是 RowShareLock 锁不会互斥 ExclusiveLock 和 AccessExclusiveLock 两种排它锁；而 Informix 的更新锁却会互斥其他任何锁（参见表 3-2 中右数第二列即“U”列的 U 锁互斥其他任何粒度的锁）。这点说明，不同的数据库，即使使用的都是封锁并发访问控制技术，但是锁的粒度不同，锁的规则也不尽相同。对于不同的数据库，需要仔细甄别。
- X：排他锁。互斥所有的操作。

这六种粒度的锁，遵循表 3-2 的相容性矩阵（一个单元格中的 Y 表示允许对另外并发的事务授予新申请的锁），在并发的任务之间，持锁者（Granted Mode，已经授予的锁）允许或排斥后发出施加锁的请求（Requested Mode，正申请的锁）。

表 3-2 Informix 锁的相容性矩阵表（不同的事务间新锁的申请）

		Granted Mode，已经授予的锁						
		N	IS	IX	S	SIX	U	X
Requested Mode 正申请的锁	IS	Y	Y	Y	Y	Y		
	IX	Y	Y	Y				
	S	Y	Y		Y			
	SIX	Y	Y					
	U	Y			Y			
	X	Y						

### 3.3 隔离级别与数据异常

Informix 支持四种隔离级别，但是这四种隔离级别不完全等同于 SQL 标准定义的隔离级别；而且，Informix 的隔离级别的实现，和封锁、日志模式紧密相关，这点和其他的数据库有着较大的不同。本节就来讨论隔离级别以及隔离级别和封锁与日志模式之间的关系。

#### 3.3.1 Informix 支持的隔离级别

Informix 的隔离级别所遵循的语法格式如图 3-3 所示，这四种隔离级别，和 SQL 标准定义的四隔离级别不尽相同，Informix 没有定义可串行化隔离级别，而 SQL 标准要求了可串行化隔离级别，且要求可串行化隔离级别可以避免幻象异常。但是，Informix 的可重复读隔离级别，就能够避免幻象异常，其具体的实现方式如下：

- ❑ 有索引的情况，只锁定一个范围，类似于 InnoDB 采用的 Next-key 之意，锁定一个间隙。这样能够避免幻象异常。
- ❑ 没有索引的情况，使用共享锁锁定相关的物理页面，根据表 3-2 的锁相容性矩阵，共享锁排斥写锁和更新锁，即此时不允许写操作的并发；而且此时锁定的对象不是元组，而是比元组范围大的物理页面，这也相当于锁定了一个大的范围，从而避免了幻象异常。
- ❑ 使用共享锁锁定相关的物理页面，共享锁排斥写锁和更新锁：意味着除了读-读并发，别的并发操作是不允许的；而结合 SS2PL，锁只在事务结束后释放，相当于实现了序列化，所以并发的调度是一个可串行化的调度，这样就能完全避免数据异常。所以，Informix 的可重复读隔离级别实际上相当于 SQL 标准的序列化隔离级别。

#### SET ISOLATION statement

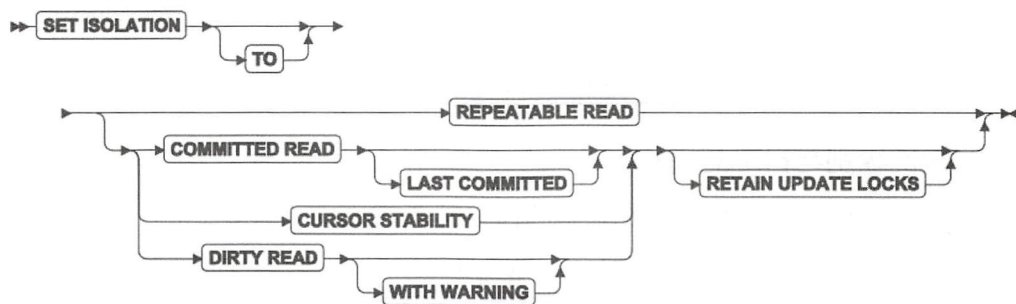


图 3-3 隔离级别语法图

对不同隔离级别，Informix 加锁和持锁、释放锁的情况，如表 3-3 所示，这个表里，使用到两个概念，一是合式事务<sup>①</sup>，二是两阶段。合式事务的概念已经很少有人提及，其含义是：

① The principle of well-formed transaction(合式事务) is defined as a transaction where the user is unable manipulate data arbitrarily, but only in constrained (limitations or boundaries) ways that preserve or ensure the integrity of the data.

```
Well Formed Transactions:
1 X-Lock an object before modifying it
2 S-Lock an object before reading it
```

而两阶段，是指整个事务封锁分为加锁阶段和解锁阶段，进入解锁阶段，即不能再施加锁。合式事务和两阶段锁保证了并发操作封锁存在的时长，结合事务结束后锁才释放的SS2PL就确保了事务内涉及的数据的一致性。

表3-3中，对于“Dirty Read”级别，读操作不加锁所以合式事务和两阶段的概念不适用；对于“Comitted Read”，读操作结束后，锁被立即释放掉，此后读其他数据，需要继续加锁，所以不符合两阶段；对于“Cursor stability”，读操作施加的共享锁，一直持有到下一次获取其他数据时才释放，而不是读后立即释放。

“Cursor stability”，不是SQL标准定义的隔离级别。当使用此隔离级别的事务利用索引<sup>①</sup>通过游标从表中查询元组时，其他并发事务不能更新或删除游标所引用的元组。但是，如果不是利用索引访问，那么其他事务可以插入新元组、更新和/或删除其他元组。

表3-3中，不同的隔离级别，解决不同的数据异常。对于SQL标准定义的三种数据异常，“Dirty Read”隔离级别不能解决SQL标准定义的脏读异常，“Comitted Read”隔离级别可以避免脏读而只读取到其他事务的已经提交的数据；“Repeatable Read”则如同SQL标准定义的可串行化，能够避免SQL标准定义的三种数据异常，也能够避免写偏序异常（参见3.3.3节）。

表 3-3 Informix 对不同隔离级别的加锁与持锁表

SQL 标准名称	Informix 定义	写操作	读操作
READ UNCOMMITTED 未提交读	Dirty Read	合式事务 / 两阶段	-/-
READ COMMITTED 已提交读	Comitted Read	合式事务 / 两阶段	合式事务 /-
	Cursor stability	合式事务 / 两阶段	合式事务 /*
REPEATABLE READ 可重复读			
SERIALIZABLE 可串行化	Repeatable Read	合式事务 / 两阶段	合式事务 / 两阶段

3.3.2 隔离级别与日志的模式

Informix 数据库的日志模式有四种，分别是：

- ❑ 无日志模式：不会记录逻辑日志，也没有事务的概念，即不能显式执行 BEGIN、ROLLBACK、COMMIT 操作。此模式下，只能使用 Dirty Read 隔离级别，所涉及的锁只会影响当前正在执行的语句。
- ❑ 缓冲日志模式：事务信息被记录在逻辑日志中，但不被立即刷出到存储介质，而

① 通过索引读取数据和直接做表扫描获取数据，对于不同隔离级别进行加锁操作是有影响的，这点需要注意。在MySQL的章节中，不特意提及这个差别，是因为InnoDB使用的是索引组织表的结构，加锁操作已经在索引上施加了。而以堆表为主的数据库系统，多提供索引，这时就需要有类似此处的区分。对于具体的问题，需要在具体的实例系统中结合锁相容性矩阵和锁粒度加以分析（不同实例系统的锁相容性矩阵和锁粒度也是不同的）。



是缓冲在日志缓存区等待日志缓存区满后执行刷出。可以显式执行 BEGIN、ROLLBACK、COMMIT 操作，适用于各种隔离级别。

- ❑ 非缓冲日志模式：事务信息被记录在逻辑日志中，但是使用该日志模式的数据库在事物提交时会立即将相关的事务信息写入磁盘，确保 WAL 机制被落实，这样才不会丢失任何已提交的事务信息。可以使用 WITH LOG 关键字在创建表时指定此日志模式，如 “CREATE DATABASE stores\_demo WITH LOG;”。
- ❑ ANSI 日志模式：使用该日志模式的数据库的缺省隔离级为 Repeatable Read，这意味着在处理的每条记录上将会被设置上共享锁。这会导致锁冲突及所等待现象的出现。

3.3.3 写偏序异常

1. 数据准备

```
CREATE TABLE dots (  
    id INT NOT NULL PRIMARY KEY,  
    color VARCHAR(20) NOT NULL  
) LOCK MODE ROW;    //对表采取行锁模式  
  
INSERT INTO dots VALUES (1, 'black');  
INSERT INTO dots VALUES (2, 'white');  
INSERT INTO dots VALUES (3, 'black');  
INSERT INTO dots VALUES (4, 'white');
```

2. 可重复读 Repeatable Read 隔离级别

对于可重复读隔离级别，并发执行如表 3-4 所示的命令。

表 3-4 隔离级别为 “REPEATABLE READ” 的并发表

Client1	Client2
BEGIN;	
SET ISOLATION REPEATABLE READ;	
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
	BEGIN;
	SET ISOLATION REPEATABLE READ;
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新失败，报告错误如下： 243: Could not position within a table (informix.dots). 113: ISAM error: the file is locked.

(续)

Client1	Client2
<pre>SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)</pre>	

从表中可以看出，行锁模式下，可重复读隔离级别，已经能够避免写偏序发生，这是因为尽管我们为表设置了行锁模式，但是 Informix 在此种情况下为 Client1 中的事务会在表上施加“HDR+SIX”锁，因此 Client2 中的 UPDATE 操作请求加锁失败，所以不能并发地执行，这样就避免了写偏序异常。

### 3.4 本章小结

本章从三个角度，简略讲述了 Informix 的事务管理方式、封锁的相关内容，以及不同隔离级别与数据异常现象的关系。这三者是事务管理和并发控制的基础。

Informix 是典型的采用 SS2PL 技术的代表，而 SS2PL 技术又是使用的最为广泛的并发访问控制技术，其他许多的并发访问控制技术都以 SS2PL 技术为基础，如 PostgreSQL、InnoDB 等在 SS2PL 技术基础上实现了 MVCC 技术，这为我们理解更多的并发控制技术提供了基础；另外，SQL 标准制定的并发控制相关的内容，原型始于封锁控制技术；所以在实例研究这一篇中，Informix 被列在了最前面。期望读者能够从中领悟技术之间的关联关系、并时刻用 Informix 来对比其他数据库的事务管理和并发访问控制技术。

## 第 4 章

# PostgreSQL 事务管理与并发控制

PostgreSQL 支持 ACID，对于不同的特性，分别使用了如下技术予以支持：

- A：原子性，通过提供事务的完整管理模型，在系统的运行过程中实现事务的提交和回滚功能，以支持运行原子性。提供 WAL 日志和恢复机制，以支持恢复原子性。
- C：一致性，通过提供基于封锁技术和 MVCC 与快照项结合的技术，以支持数据在运行期间数据被并发修改情况下的运行一致（事务故障通过回滚实现运行一致）、通过 WAL 日志和恢复机制以支持故障发生后的一致（系统故障、介质故障）。
- I：隔离性，通过“SET TRANSACTION ISOLATION LEVEL...”语句来设置满足 ANSI SQL 标准规定的四种隔离级别中的三种（PostgreSQL 不支持游标稳定隔离级别），实现并发事务的不同场景的隔离。通过封锁技术、以快照和多版本构成的 MVCC 技术实现并发事务对同一个数据项的读写隔离。
- D：持久性，通过预写日志（WAL）、恢复等技术实现了数据的持久化存储。

如下，将对事务管理和并发控制技术进行讨论，这些技术涉及了 A、C、I 这几个特性。

## 4.1 PostgreSQL 事务操作

PostgreSQL 提供了完整的事务和子事务管理方面的相关操作，用户可以主动开启一个事务或子事务，主动提交或回滚事务及子事务。如果不主动控制事务，则 PostgreSQL 采用了自动控制事务的方式。



### 4.1.1 开始事务

用户可以主动开启一个事务，使用的命令如下。

#### 1. 开启主事务块

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
transaction_mode:
    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

开启一个新的事务，可以同时指定事务的隔离级别，指定事务的读写属性等。如果是只读的事务（READ ONLY）则可以为只读事务指定是否延迟（DEFERRABLE）的特性，这是 PostgreSQL 在用 SSI（可串行化快照隔离）技术实现可串行化隔离级别时，对只读事务所作的一个优化，详情可以参见 9.4.2 节。

PostgreSQL 的隔离级别只能在一个事务内部设定，不能在会话层设定然后对所有的事务有效。因此隔离级别的设置被绑定在了开启事务的语句上。当然，开启一个事务后，还可以通过“SET TRANSACTION ISOLATION LEVEL”语句单独设置事务的隔离级别。

除了 BEGIN 关键字可以开启一个事务块外，START 关键字也有同等功效。

#### 2. 开启子事务块

PostgreSQL 实现了平板事务模型，也提供了保存点功能。定义一个保存点的方式如下：

```
SAVEPOINT savepoint_name
```

在 PostgreSQL 内部，使用保存点来模拟子事务的方式实现的是带有保存点的平板事务模型，不是真正的嵌套事务模型。

### 4.1.2 提交事务

用户可以使用 COMMIT 命令直接提交一个事务，以确保事务块中的写操作生效。事务一旦提交，事务的生命周期就全部结束，包括通过保存点模拟的“子事务”也随着事务的提交而提交。保存点之前的操作，是不能独立提交的。所以 PostgreSQL 只提供了如下的语句提交整个事务。

```
COMMIT [ WORK | TRANSACTION ]
```

### 4.1.3 回滚事务

事务可以被整体回滚，通过保存点模拟的“子事务”也可以被局部回滚。

#### 1) 回滚主事务块：

```
ROLLBACK [ WORK | TRANSACTION ]
```

## 2) 回滚子事务块:

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

回滚的保存点所限定的范围是局部的，不影响事务块内的其他部分。

## 4.1.4 XA 事务

XA 事务的管理，通过如下三条语句进行。

```
PREPARE TRANSACTION transaction_id //开始一个XA事务
COMMIT PREPARED transaction_id      //提交一个XA事务
ROLLBACK PREPARED transaction_id    //回滚一个XA事务
```

XA 事务用法的一个示例如下:

```
CREATE TABLE t(col INT);           //数据准备

BEGIN;                               //开始一个事务
  INSERT INTO t VALUES(1);
  INSERT INTO t VALUES(2);
  PREPARE TRANSACTION 'x1';         //标识之前的语句已经准备好提交，即从BEGIN到本语句之间的
                                     语句是一个分布式的事务块
  INSERT INTO t VALUES(3);         //不包括在PREPARE之内的另外一个子事务块
COMMIT;                             //提交了插入值为“3”的语句

SELECT * FROM t;                    //查询到的值是3
COMMIT PREPARED 'x1';               //提交“PREPARE TRANSACTION 'x1';”语句之前的语句，
                                     即插入值为“1”和“2”的语句
SELECT col FROM t;                  //查询到的值是1、2、3
```

## 4.1.5 自动控制事务

如果不使用 BEGIN 语句主动开启一个事务块，PostgreSQL 会隐式地自动开启一个事务块，单个 SQL 语句为一个事务，执行正确则自动提交，执行失败则自动回滚。

DDL 语句，如果处于一个事务块内部，不会主动先提交事务 DDL 语句之前执行的语句，并隐式开启一个新的事务，而是随着事务块的提交或回滚而提交或回滚。这点和 MySQL 的 InnoDB 不同。示例如下:

```
postgres=# SELECT * FROM t1;         //t1表中只有2行数据
 a | b | c
---+---+---
 1 | 1 | 1
 2 | 2 | 2
(2 rows)
```



```

postgres=# BEGIN;                                //开启一个事务块
BEGIN
postgres=# DELETE FROM t1 WHERE a=2;             //删除一条数据
DELETE 1
postgres=# CREATE TABLE t3(a INT);              //创建一个t3表成功
CREATE TABLE
postgres=# ROLLBACK;                             //回滚事务，则创建表的DDL被回滚掉，而创建表的DDL
                                                没有隐式提交之前的删除操作

ROLLBACK
postgres=# SELECT * FROM t1;                     //创建表的DDL没有隐式提交之前的删除操作，数据依旧是2行
 a | b | c
---+---+---
 1 | 1 | 1
 2 | 2 | 2
(2 rows)

postgres=# SELECT * FROM t3;                     //回滚事务，则创建表的DDL被回滚掉，所以t3表不存在
ERROR:  relation "t3" does not exist
LINE 1: SELECT * FROM t3;

```

如果一个事务块内，一条 SQL 执行失败，PostgreSQL 报告错误，则这个事务块会自动回滚。这是因为 PostgreSQL 的事务模型实现的方式，决定了一旦事务内部发生错误，系统自动隐含地调用了回滚函数处理错误，详情请参见 7.3.4 节的“3 隐式的事务回滚”。

## 4.2 SQL操作与锁

PostgreSQL 对于各种 SQL 操作，施加的事务锁是不同的，在第 8 章对事务锁有着详细的讨论，本节从示例的角度，分析 PostgreSQL 执行 SQL 加锁的初步情况。

### 4.2.1 锁的研究准备

创建视图，用于查询数据对象上加锁的情况：

```

CREATE VIEW active_locks AS
//创建视图，用于查询数据对象上加锁的情况。PostgreSQL V9.6验证通过
SELECT pg_class.relname, pg_locks.locktype, pg_locks.database,
       pg_locks.relation, pg_locks.page, pg_locks.tuple, pg_locks.virtualtransaction,
       pg_locks.pid, pg_locks.mode, pg_locks.granted,
       CASE
         WHEN virtualxid IS NOT NULL AND transactionid IS NOT NULL
         THEN virtualxid || ' ' || transactionid
         WHEN virtualxid::text IS NOT NULL

```



```

        THEN virtualxid
        ELSE transactionid::text
    END AS xid_lock
FROM pg_locks LEFT JOIN pg_class ON pg_locks.relation = pg_class.oid
WHERE relname !~ '^pg_' and relname <> 'active_locks';

SELECT * FROM active_locks; //查询视图

```

创建的视图基于 PostgreSQL 的系统表 `pg_locks`，主要列对象的含义如表 4-1 所示。

表 4-1 PostgreSQL 的系统表 `pg_locks` 主要列对象的含义

名字	类型	描述
locktype	text	可锁定对象的类型： relation、extend、page、tuple、transactionid、object、userlock
database	oid	对象所在的数据库的 OID，如果对象是共享对象，值是零； 如果对象是一个事务 ID，值是 NULL
relation	oid	关系的 OID，如果对象不是关系，也不是关系的一部分，值为 NULL
page	integer	关系内部的页面编号，如果对象不是元组页不是关系页，值为 NULL
tuple	smallint	页面里面的元组编号，如果对象不是元组，值为 NULL
transactionid	xid	事务的 ID，如果对象不是事务，值是 NULL
classid	oid	包含该对象的系统表的 OID，如果对象不是普通数据库对象，值为 NULL
objid	oid	对象在其系统表内的 OID，如果对象不是普通数据库对象，值为 NULL
objsubid	smallint	对于表的一个字段，这是字段编号（classid 和 objid 指向表自身）； 对于其他对象类型，值是零； 如果这个对象不是普通数据库对象，值为 NULL
transaction	xid	持有此锁或者在等待此锁的事务的 ID
pid	integer	持有或者等待这个锁的服务器进程的进程 ID； 如果锁是被一个准备好的事务持有的，值为 NULL
mode	text	这个进程持有的或者是期望的锁模式
granted	boolean	如果持有锁，为真；如果等待锁，为假

创建表如下：

```

CREATE TABLE parent (
    pid INT PRIMARY KEY,
    a1 INT, b1 INT);

CREATE TABLE child (
    sid INT PRIMARY KEY,
    parent_id INT NOT NULL,
    a2 INT, b INT, CONSTRAINT child_parent_fk FOREIGN KEY (parent_id)
    REFERENCES parent(pid)
);

```

### 4.2.2 INSERT 操作触发的锁

例 1，隔离级别为可串行化：

```
BEGIN; //利用2PC机制，释放锁发生在事务提交或回滚阶段，所以每个命令执行后加锁的情况可以被方便查知
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO parent (pid, a1, b1) VALUES (1, 1, 1);
SELECT * FROM active_locks;
INSERT INTO parent (pid, a1, b1) VALUES (2, 2, 2);
SELECT * FROM active_locks;
COMMIT;
```

每次查询，都可以得到类似如下的结果，表明在表上增加了一个“RowExclusiveLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted	xid_lock
parent	relation	12401	16389		2/11	12028	RowExclusiveLock	t		

例 2，隔离级别为已提交读：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO parent (pid, a1, b1) VALUES (3, 3, 3);
SELECT * FROM active_locks;
COMMIT;
```

查询，可以得到类似如下的结果，表明在表上增加了一个“RowExclusiveLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted	xid_lock
parent	relation	12401	16389		2/13	12028	RowExclusiveLock	t		

通过这两个例子，可以看出，插入语句所加的锁“RowExclusiveLock”操作与隔离级别没有关系。

### 4.2.3 SELECT 操作触发的锁

例 1，隔离级别为可串行化：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM parent;
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，得到类似如下的结果，表明在表上增加了一个表级锁“AccessShareLock”和谓词锁“SIReadLock”，在索引 parent\_pkey 上增加了一个



“AccessShareLock” 锁:

```

relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
parent_pkey | relation | 12401 | 16392 | | | 2/14 | 12028 | AccessShareLock | t
parent | relation | 12401 | 16389 | | | 2/14 | 12028 | AccessShareLock | t
parent | relation | 12401 | 16389 | | | 2/14 | 12028 | SIReadLock | t

```

例 2, 隔离级别为可串行化, 但查询使用索引并读取表数据:

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM parent WHERE pid=2; // “pid=2” 指明要使用索引, 但 “*” 表明要读取表数据
SELECT * FROM active_locks;
COMMIT;

```

查询 active\_locks 视图, 可以得到类似如下的结果, 表明在表上增加了一个表级锁 “AccessShareLock” 在元组上增加了谓词锁 “SIReadLock”, 在索引 parent\_pkey 上增加了一个 “AccessShareLock” 锁且在索引页上增加了一个谓词锁 “SIReadLock”:

```

relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
parent_pkey | relation | 12401 | 16392 | | | 2/18 | 12028 | AccessShareLock | t
parent | relation | 12401 | 16389 | | | 2/18 | 12028 | AccessShareLock | t
parent | tuple | 12401 | 16389 | 0 | 4 | 2/18 | 12028 | SIReadLock | t
parent_pkey | page | 12401 | 16392 | 1 | | 2/18 | 12028 | SIReadLock | t

```

例 3, 隔离级别为可串行化, 但查询只在索引上进行:

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT pid FROM parent WHERE pid=2; // “pid=2” 指明要使用索引, 但 “pid” 表明不会
读取表数据, 只需要读取索引即可
SELECT * FROM active_locks;
COMMIT;

```

查询 active\_locks 视图, 得到类似如下的结果, 表明在表上增加了一个表级锁 “AccessShareLock” 在元组上增加了谓词锁 “SIReadLock”, 在索引 parent\_pkey 上增加了一个 “AccessShareLock” 锁且在索引页上增加了一个谓词锁 “SIReadLock”:

```

relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
parent_pkey | relation | 12401 | 16392 | | | 2/17 | 12028 | AccessShareLock | t
parent | relation | 12401 | 16389 | | | 2/17 | 12028 | AccessShareLock | t
parent_pkey | page | 12401 | 16392 | 1 | | 2/17 | 12028 | SIReadLock | t
parent | tuple | 12401 | 16389 | 0 | 4 | 2/17 | 12028 | SIReadLock | t

```



例 4，隔离级别为已提交读：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM parent;
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上和索引上分别增加了一个“RowExclusiveLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/15	12028	AccessShareLock	t
parent	relation	12401	16389			2/15	12028	AccessShareLock	t

例 5，隔离级别为可串行化，但查询使用索引并读取表数据：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM parent WHERE pid=2; // “pid=2” 指明要使用索引，但 “*” 表明要读取表数据
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁“AccessShareLock”，在索引 parent\_pkey 上增加了一个“AccessShareLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/20	12028	AccessShareLock	t
parent	relation	12401	16389			2/20	12028	AccessShareLock	t

例 6，隔离级别为可串行化，但查询只在索引上进行：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT pid FROM parent WHERE pid=2; // “pid=2” 指明要使用索引，但 “pid” 表明不会读取表数据，只需要读取索引即可
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁“AccessShareLock”，在索引 parent\_pkey 上增加了一个“AccessShareLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/21	12028	AccessShareLock	t
parent	relation	12401	16389			2/21	12028	AccessShareLock	t

对例 1 ~ 例 6 进行对比，如表 4-2 所示，可以看出，查询语句所加的锁与隔离级别有关系，可串行化隔离级别使用谓词锁，其加锁多于已提交读。

表 4-2 查询语句加锁情况与隔离级别的关系

	表 (relation)	元组	索引	索引页	特点	共同点
1	AccessShareLock SIReadLock		AccessShareLock		不读取索引，但在索引上加共享锁	可串行化 隔离级别
2	AccessShareLock	SIReadLock	AccessShareLock	SIReadLock	读取索引和表数据，表和索引都加锁	
3	AccessShareLock	SIReadLock	AccessShareLock	SIReadLock	只读取索引的数据，但同时锁表和索引	
4	AccessShareLock		AccessShareLock		不读取索引，但在索引上加共享锁	可重复读 隔离级别
5	AccessShareLock		AccessShareLock		读取索引和表数据，表和索引都加锁	
6	AccessShareLock		AccessShareLock		只读取索引的数据，但同时锁表和索引	

4.2.4 SELECT FOR UPDATE 操作触发的锁

例 1，隔离级别为可串行化：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM parent FOR UPDATE;
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，得到类似如下的结果，表明在表上增加了一个表级锁“RowShareLock”（这一点与单纯的 SELECT 操作加“AccessShareLock”锁不同）和谓词锁“SIReadLock”，在索引 parent\_pkey 上增加了一个“AccessShareLock”锁：

```
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
parent_pkey | relation | 12401 | 16392 | | | 2/37 | 12028 | AccessShareLock | t
parent | relation | 12401 | 16389 | | | 2/37 | 12028 | RowShareLock | t
parent | relation | 12401 | 16389 | | | 2/37 | 12028 | SIReadLock | t
```

例 2，隔离级别为可串行化，但查询使用索引并读取表数据：



```

BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM parent WHERE pid=2 FOR UPDATE; // “pid=2”指明要使用索引，但“*”表明
要读取表数据
SELECT * FROM active_locks;
COMMIT;

```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁 “AccessShareLock” 在元组上增加了谓词锁 “SIReadLock”，在索引 parent\_pkey 上增加了一个 “AccessShareLock” 锁且在索引页上增加了一个谓词锁 “SIReadLock”：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/38	12028	AccessShareLock	t
parent	relation	12401	16389			2/38	12028	RowShareLock	t
parent	tuple	12401	16389	0	9	2/38	12028	SIReadLock	t
parent_pkey	page	12401	16392	1		2/38	12028	SIReadLock	t

例 3，隔离级别为可串行化，但查询只在索引上进行：

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT pid FROM parent WHERE pid=2 FOR UPDATE; // “pid=2”指明要使用索引，但“pid”
表明不会读取表数据，只需要读取索引即可
SELECT * FROM active_locks;
COMMIT;

```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁 “RowShareLock” 在元组上增加了谓词锁 “SIReadLock”，在索引 parent\_pkey 上增加了一个 “AccessShareLock” 锁且在索引页上增加了一个谓词锁 “SIReadLock”：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/39	12028	AccessShareLock	t
parent	relation	12401	16389			2/39	12028	RowShareLock	t
parent	tuple	12401	16389	0	9	2/39	12028	SIReadLock	t
parent_pkey	page	12401	16392	1		2/39	12028	SIReadLock	t

例 4，隔离级别为已提交读：

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM parent FOR UPDATE;
SELECT * FROM active_locks;
COMMIT;

```

查询 active\_locks 视图，得到类似如下的结果，表明在表上和索引上分别增加了一个 “RowShareLock” 锁和 “AccessShareLock” 锁：



relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/40	12028	AccessShareLock	t
parent	relation	12401	16389			2/40	12028	RowShareLock	t

例 5，隔离级别为可串行化，但查询使用索引并读取表数据：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM parent WHERE pid=2 FOR UPDATE; // “pid=2” 指明要使用索引，但 “*” 表明
要读取表数据
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁 “RowShareLock”，在索引 parent\_pkey 上增加了一个 “AccessShareLock” 锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/41	12028	AccessShareLock	t
parent	relation	12401	16389			2/41	12028	RowShareLock	t

例 6，隔离级别为可串行化，但查询只在索引上进行：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT pid FROM parent WHERE pid=2 FOR UPDATE; // “pid=2” 指明要使用索引，但 “pid”
表明不会读取表数据，只需要读取索引即可
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁 “RowShareLock”，在索引 parent\_pkey 上增加了一个 “AccessShareLock” 锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/42	12028	AccessShareLock	t
parent	relation	12401	16389			2/42	12028	RowShareLock	t

例 7，隔离级别为可串行化，但查询使用普通列并读取表数据：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM parent WHERE b1=2 FOR UPDATE; // “b1=2” 指明要使用普通列，但 “*”
表明要读取表数据
SELECT * FROM active_locks;
COMMIT;
```

查询 active\_locks 视图，得到类似如下的结果，表明在表上增加了一个表级锁

“RowShareLock”在元组上增加了谓词锁“SIReadLock”，在索引parent\_pkey上增加了一个“AccessShareLock”锁，与例2相比，没有在元组上加锁，直接在表上加了“RowShareLock”这样的排它锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation		12401	16392		2/2	6628	AccessShareLock	t
parent	relation		12401	16389		2/2	6628	RowShareLock	t
parent	relation		12401	16389		2/2	6628	SIReadLock	t

对例1～例7进行对比如表4-3所示，可以看出，带有FOR UPDATE操作的查询语句所加的锁与隔离级别有关系，可串行化隔离级别使用谓词锁，其加锁多于已提交读。而且，与单纯的SELECT操作的查询语句相比，带有FOR UPDATE操作的查询语句使用了“RowShareLock”锁，而单纯的SELECT操作的查询语句使用了“AccessShareLock”锁，后者加锁的级别低于前者。

表 4-3 带有 FOR UPDATE 的查询语句加锁情况与隔离级别的关系表

	表 (relation)	元组	索引	索引页	特点	共同点
1	RowShareLock SIReadLock		AccessShareLock		不读取索引，但在索引上加共享锁	可串行化 隔离级别
2	RowShareLock	SIReadLock	AccessShareLock	SIReadLoc	读取索引和表数据，表和索引都加锁	
3	RowShareLock	SIReadLock	AccessShareLock	SIReadLock	只读取索引的数据，但同时锁表和索引	
4	RowShareLock		AccessShareLock		不读取索引，但在索引上加共享锁	可重复读 隔离级别
5	RowShareLock		AccessShareLock		读取索引和表数据，表和索引都加锁	
6	RowShareLock		AccessShareLock		只读取索引的数据，但同时锁表和索引	
7	RowShareLock SIReadLock		AccessShareLock		读取表数据，表和索引都加锁，表上不仅加排它锁，还有谓词锁，元组没有加锁	

4.2.5 UPDATE 操作触发的锁

例 1，隔离级别为可串行化：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE parent SET b1=10;
SELECT * FROM active_locks;
ROLLBACK;
```



查询 `active_locks` 视图，得到类似如下的结果，表明在表上增加了一个表级锁 “`RowExclusiveLock`” 和谓词锁 “`SIReadLock`”，在索引 `parent_pkey` 上增加了一个 “`RowExclusiveLock`” 锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392		2/22	12028		RowExclusiveLock	t
parent	relation	12401	16389		2/22	12028		RowExclusiveLock	t
parent	relation	12401	16389		2/22	12028		SIReadLock	t

例 2，隔离级别为可串行化，但查询使用索引并读取表数据：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE parent SET b1=10 WHERE pid=2; // “pid=2” 指明要使用索引，但 “b1=10” 表明
要读取表数据
SELECT * FROM active_locks;
ROLLBACK;
```

查询 `active_locks` 视图，得到类似如下的结果，表明在表上增加了一个表级锁 “`RowExclusiveLock`”，在索引 `parent_pkey` 上增加了一个 “`RowExclusiveLock`” 锁且在索引页上增加了一个谓词锁 “`SIReadLock`”：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392		2/28	12028		RowExclusiveLock	t
parent	relation	12401	16389		2/28	12028		RowExclusiveLock	t
parent_pkey	page	12401	16392	1	2/28	12028		SIReadLock	t

例 3，隔离级别为可串行化，但查询只在索引上进行：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE parent SET pid=10 WHERE pid=2; // “pid=2” 指明要使用索引，但 “pid” 表明不会
读取表数据，只需要读取索引即可
SELECT * FROM active_locks;
ROLLBACK;
```

查询 `active_locks` 视图，得到类似如下的结果，表明在表上增加了一个表级锁 “`RowShareLock`” 和 “`RowExclusiveLock`”，在索引 `parent_pkey` 上增加了三个锁包括谓词锁 “`SIReadLock`”；另外，在子表 `child` 和子表的索引上，增加了一个谓词锁 “`SIReadLock`” 和 “`RowShareLock`” 锁且在子表的索引页上增加了 “`AccessShareLock`” 锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
child_pkey	relation	12401	16397		2/29	12028		AccessShareLock	t
child	relation	12401	16394		2/29	12028		RowShareLock	t



parent_pkey		relation		12401		16392				2/29		12028		AccessShareLock		t
parent_pkey		relation		12401		16392				2/29		12028		RowExclusiveLock		t
parent		relation		12401		16389				2/29		12028		RowShareLock		t
parent		relation		12401		16389				2/29		12028		RowExclusiveLock		t
child		relation		12401		16394				2/29		12028		SIReadLock		t
parent_pkey		page		12401		16392		1		2/29		12028		SIReadLock		t

例 4，隔离级别为已提交读：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE parent SET b1=10;
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，得到类似如下的结果，表明在表上和索引上分别增加了一个“RowExclusiveLock”锁：

relname		locktype		database		relation		page		tuple		virtualtransaction		pid		mode		granted
parent_pkey		relation		12401		16392				2/30		12028		RowExclusiveLock		t		
parent		relation		12401		16389				2/30		12028		RowExclusiveLock		t		

例 5，隔离级别为可串行化，但查询使用索引并读取表数据：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE parent SET b1=10 WHERE pid=2;
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，得到类似如下的结果，表明在表和索引 parent\_pkey 上分别增加了一个“RowExclusiveLock”锁：

relname		locktype		database		relation		page		tuple		virtualtransaction		pid		mode		granted
parent_pkey		relation		12401		16392				2/31		12028		RowExclusiveLock		t		
parent		relation		12401		16389				2/31		12028		RowExclusiveLock		t		

例 6，隔离级别为可串行化，但查询只在索引上进行：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE parent SET pid=10 WHERE pid=2;
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了表级锁“RowShareLock”和“RowExclusiveLock”，在索引 parent\_pkey 上增加了“AccessShare-

Lock”和“RowExclusiveLock”锁；在子表child和子表的索引上分别增加了“RowShareLock”和“AccessShareLock”锁：

```

relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
child_pkey | relation | 12401 | 16397 | | | 2/32 | 12028 | AccessShareLock | t
child | relation | 12401 | 16394 | | | 2/32 | 12028 | RowShareLock | t
parent_pkey | relation | 12401 | 16392 | | | 2/32 | 12028 | AccessShareLock | t
parent_pkey | relation | 12401 | 16392 | | | 2/32 | 12028 | RowExclusiveLock | t
parent | relation | 12401 | 16389 | | | 2/32 | 12028 | RowShareLock | t
parent | relation | 12401 | 16389 | | | 2/32 | 12028 | RowExclusiveLock | t

```

对例1～例6进行对比，如表4-4所示，可以看出，更新语句所加的锁与隔离级别有关系，可串行化隔离级别使用谓词锁，其加锁多于已提交读。

表 4-4 更新语句加锁情况与隔离级别的关系表

	表 (relation)	元组	索引	索引页	子表	子表索引	特点	共同点
1	RowExclusive Lock SIReadLock		RowExclusive Lock				不读取索引，但在索引上加共享锁	可串行化隔离级别
2	RowExclusive Lock		RowExclusive Lock	SIReadLock			读取索引和表数据，表和索引都加锁	
3	RowShareLock RowExclusive Lock		AccessShare Lock RowExclusive Lock	SIReadLock	RowShare Lock	AccessShare Lock	只读取索引的数据，但同时锁表和索引	
4	RowExclusive Lock		RowExclusive Lock				不读取索引，但在索引上加共享锁	可重复读隔离级别
5	RowExclusive Lock		RowExclusive Lock				读取索引和表数据，表和索引都加锁	
6	RowShareLock RowExclusive Lock		AccessShare Lock RowExclusive Lock		RowShare Lock	AccessShare Lock 如果子表数据被修改会加排它锁	只读取索引的数据，但同时锁表和索引	

#### 4.2.6 DELETE 操作触发的锁

例1，隔离级别为可串行化：

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
DELETE FROM parent WHERE b1=10;

```



```
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁 “RowExclusiveLock” 和谓词锁 “SIReadLock”，在索引 parent\_pkey 上增加了一个 “RowExclusiveLock” 锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/33	12028	RowExclusiveLock	t
parent	relation	12401	16389			2/33	12028	RowExclusiveLock	t
parent	relation	12401	16389			2/33	12028	SIReadLock	t

例 2，隔离级别为可串行化，但查询使用索引并读取表数据：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
DELETE FROM parent WHERE pid=2; // “pid=2” 指明要使用索引
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了表级锁 “RowShareLock” 和 “RowExclusiveLock”，在索引 parent\_pkey 上增加了 “AccessShareLock” 和 “RowExclusiveLock” 锁，且在索引页上增加了一个谓词锁 “SIReadLock”；另外，在子表上增加了 “RowShareLock” 和谓词锁 “SIReadLock”，在子表的索引上增加了 “AccessShareLock”：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
child_pkey	relation	12401	16397			2/34	12028	AccessShareLock	t
child	relation	12401	16394			2/34	12028	RowShareLock	t
parent_pkey	relation	12401	16392			2/34	12028	AccessShareLock	t
parent_pkey	relation	12401	16392			2/34	12028	RowExclusiveLock	t
parent	relation	12401	16389			2/34	12028	RowShareLock	t
parent	relation	12401	16389			2/34	12028	RowExclusiveLock	t
parent_pkey	page	12401	16392	1		2/34	12028	SIReadLock	t
child	relation	12401	16394			2/34	12028	SIReadLock	t

例 3，隔离级别为读已提交：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
DELETE FROM parent WHERE b1=10;
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上和索引上分别增加了一个 “RowExclusiveLock” 锁：



```

relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
parent_pkey | relation | 12401 | 16392 | | | 2/35 | 12028 | RowExclusiveLock | t
parent      | relation | 12401 | 16389 | | | 2/35 | 12028 | RowExclusiveLock | t

```

例 4，隔离级别为可串行化，但查询使用索引并读取表数据：

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
DELETE FROM parent WHERE pid=2;
SELECT * FROM active_locks;
ROLLBACK;

```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了表级锁“RowShareLock”和“RowExclusiveLock”，在索引 parent\_pkey 上增加了“AccessShareLock”和“RowExclusiveLock”锁；在子表 child 和子表的索引上分别增加了“RowShareLock”和“AccessShareLock”锁：

```

relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
child_pkey | relation | 12401 | 16397 | | | 2/36 | 12028 | AccessShareLock | t
child      | relation | 12401 | 16394 | | | 2/36 | 12028 | RowShareLock     | t
parent_pkey | relation | 12401 | 16392 | | | 2/36 | 12028 | AccessShareLock | t
parent_pkey | relation | 12401 | 16392 | | | 2/36 | 12028 | RowExclusiveLock | t
parent      | relation | 12401 | 16389 | | | 2/36 | 12028 | RowShareLock     | t
parent      | relation | 12401 | 16389 | | | 2/36 | 12028 | RowExclusiveLock | t

```

对例 1 ~ 例 4 进行对比，如表 4-5 所示，可以看出，删除语句所加的锁与隔离级别有关系，可串行化隔离级别使用谓词锁，其加锁多于已提交读。

表 4-5 删除语句加锁情况与隔离级别的关系

	表 (relation)	元组	索引	索引页	子表	子表索引	特点	共同点
1	RowExclusive Lock SIReadLock		RowExclusive Lock				不读取索引，但在索引上加排它锁	可串行化隔离级别
2	RowShareLock RowExclusive Lock		AccessShare Lock RowExclusive Lock	SIRaad Lock	RowShare Lock	AccessShar eLock	读取索引和表数据，表和索引都加锁	
3	RowExclusive Lock		RowExclusive Lock				不读取索引，但在索引上加排它锁	可重复读隔离级别
4	RowShareLock RowExclusive Lock		AccessShare Lock RowExclusive Lock		RowShare Lock	AccessShare Lock	读取索引和表数据，表和索引都加锁	

### 4.2.7 ANALYZE 操作触发的锁

例 1，隔离级别为可串行化：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
ANALYZE parent;
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁“ShareUpdateExclusiveLock”，在索引 parent\_pkey 上增加了一个“AccessShareLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/7	10460	AccessShareLock	t
parent	relation	12401	16389			2/7	10460	ShareUpdateExclusiveLock	t

例 2，隔离级别为已提交读：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
ANALYZE parent;
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明 ANALYZE 操作在表上和索引增加锁与不同的隔离级别没有关系：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
parent_pkey	relation	12401	16392			2/7	10460	AccessShareLock	t
parent	relation	12401	16389			2/7	10460	ShareUpdateExclusiveLock	t

### 4.2.8 CREATE INDEX 操作触发的锁

例 1，隔离级别为可串行化：

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
CREATE INDEX p_b1 ON parent (b1);
SELECT * FROM active_locks;
ROLLBACK;
```

查询 active\_locks 视图，可以得到类似如下的结果，表明在表上增加了一个表级锁“ShareLock”，在索引 p\_b1 上增加了一个“AccessShareLock”锁：

relname	locktype	database	relation	page	tuple	virtualtransaction	pid	mode	granted
---------	----------	----------	----------	------	-------	--------------------	-----	------	---------



```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
parent | relation | 12401 | 16389 | | | 2/9 | 10460 | ShareLock | t
p_b1   | relation | 12401 | 24576 | | | 2/9 | 10460 | AccessExclusiveLock | t

```

例 2，隔离级别为已提交读：

```

BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
CREATE INDEX p_b1 ON parent (b1);
SELECT * FROM active_locks;
ROLLBACK;

```

查询 active\_locks 视图，可以得到类似如下的结果，表明 CREATE INDEX 操作在表上和索引增加锁与不同的隔离级别没有关系：

```

relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
parent  | relation | 12401   | 16389   | | | 2/10 | 10460 | ShareLock | t
p_b1    | relation | 12401   | 24577   | | | 2/10 | 10460 | AccessExclusiveLock | t

```

## 4.2.9 CREATE TRIGGER 操作触发的锁

从创建触发器的代码可以看出，CREATE TRIGGER 操作在表上增加锁与不同的隔离级别没有关系，所加的锁为 “ShareRowExclusiveLock”：

```

CreateTrigger(CreateTrigStmt *stmt, const char *queryString, Oid relOid,
Oid refRelOid, Oid constraintOid, Oid indexOid, bool isInternal)
{...
    if (OidIsValid(relOid))
        rel = heap_open(relOid, ShareRowExclusiveLock);
    else
        rel = heap_openrv(stmt->relation, ShareRowExclusiveLock);
    ...
}

```

从创建主外键的代码可以看出，创建主外键操作在主键表上增加锁与不同的隔离级别没有关系，所加的锁为 “ShareRowExclusiveLock”：

```

ATAddForeignKeyConstraint(AlteredTableInfo *tab, Relation rel,
Constraint *fkconstraint, LOCKMODE lockmode)
{...
    // Grab ShareRowExclusiveLock on the pk table, so that someone doesn't delete
    rows out from under us.
    if (OidIsValid(fkconstraint->old_pktable_oid))
        pkrel = heap_open(fkconstraint->old_pktable_oid, ShareRowExclusiveLock);
    else
        pkrel = heap_openrv(fkconstraint->pktable, ShareRowExclusiveLock);
    ...
}

```



## 4.2.10 锁的相关参数

PostgreSQL 提供如下参数，用以控制封锁管理。

```
postgres=# SELECT name,setting,unit,category FROM pg_settings WHERE name
LIKE '%lock%' AND name NOT LIKE '%block%';
```

name	setting	unit	category
deadlock_timeout	1000	ms	Lock Management
lock_timeout	0	ms	Client Connection Defaults / Statement Behavior
log_lock_waits	off		Reporting and Logging / What to Log
max_locks_per_transaction	64		Lock Management
max_pred_locks_per_transaction	64		Lock Management

(5 rows)

- ❑ **deadlock\_timeout**：死锁检测延时的控制参数，单位是毫秒，默认值为 1 秒。死锁检测的过程非常耗费时间，增加死锁检测延时控制参数，以延缓死锁检测过程的执行。
- ❑ **lock\_timeout**：锁等待超时的控制参数，单位是毫秒，默认值是 0 秒，表示不进行超时控制。如果设置了此变量，表示事务运行时间超过其值时，需要被回滚。
- ❑ **log\_lock\_waits**：锁等待如果超时，是否记录一个日志信息。默认值为 OFF，表示关闭此开关，不记录一个锁等待超时信息。如果此开关打开，这样的信息很多，就需要追查系统产生锁等待的原因。
- ❑ **max\_locks\_per\_transaction**：平均每个事务的最大允许施加的锁（不包括谓词锁）的个数。此参数只是一个平均值，不是单个事务被允许施加的最大锁个数的上限。默认值是 64。全系统所有的锁的总个数是： $\text{max\_locks\_per\_transaction} * (\text{max\_connections} + \text{max\_prepared\_transactions})$ 。此参数只是一个平均值，不是单个事务被允许施加的最大锁个数的上限。
- ❑ **max\_pred\_locks\_per\_transaction**：平均每个事务的最大允许施加的谓词锁的个数。此参数只是一个平均值，不是单个事务被允许施加的最大谓词锁个数的上限。默认值是 64。全系统所有的锁的总个数是： $\text{max\_pred\_locks\_per\_transaction} * (\text{max\_connections} + \text{max\_prepared\_transactions})$ 。

## 4.3 隔离级别与数据异常

### 4.3.1 SQL 标准定义的三种读异常

PostgreSQL 支持 ANSI SQL 标准定义的四种隔离级别，分别为：可串行为 SERIALIZABLE、可重复读 REPEATABLE READ、已提交读 READ COMMITTED 和未提交读 READ UNCOMMITTED。其中，REPEATABLE READ 是默认值。设置隔离级别需要在一个事务块内，通过类似“SET TRANSACTION ISOLATION LEVEL SERIALIZABLE”语句来完成。

这四种隔离级别，分别用于解决三种读异常现象和写偏序异常。符合 2.2.1 节我们讨论的基于锁的封锁并发控制技术。但是，因为 PostgreSQL 使用了 MVCC 并发控制技术，使得 PostgreSQL 在不同隔离级别下的表现与基于锁的 Informix 不完全相同，具体情况可以参见如下内容，分别验证了四种隔离级别的多事务并发情况。

首先，构造环境如下：

```
CREATE TABLE bluesea (c1 INT, c2 VARCHAR(20), c3 INT, PRIMARY KEY(c1));
```

其次，分别启动两个客户端，Client1 和 Client2，假定 Client1 为主事务端，在此端改变隔离级别。

### 1. 可串行化隔离级别

对于可串行化隔离级别，并发执行如下命令，如表 4-6 所示。

表 4-6 隔离级别为“SERIALIZABLE”的并发表

Client1	Client2
BEGIN;	
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- (0 rows)	
	INSERT INTO bluesea values (1,'Pg-01',1); 执行成功，插入操作没有受到 Client1 的影响
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- (0 rows) 隔离级别为“SERIALIZABLE”，不存在幻象	

在 Client2 中执行如下查询锁情况的 SQL，得到这两个并发事务的加锁情况如下。其中，锁“RowExclusiveLock”是 Client2 是施加的，其余的锁是 Client1 施加的。

```
postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted
|xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
bluesea      | relation |          | 12401    | 57344 |      | 3/3 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation |          | 12401    | 57347 |      | 2/5 | 9164 | AccessShareLock  | t |
bluesea      | relation |          | 12401    | 57344 |      | 2/5 | 9164 | AccessShareLock  | t |
bluesea_pkey | relation |          | 12401    | 57347 |      | 2/5 | 9164 | SIReadLock       | t |
(4 rows)
```

可以看出，Client1 上的 AccessShareLock 锁不排斥 Client2 想要施加的 RowExclusiveLock 锁，致使 Client2 可以执行插入操作，但是 Client1 第二次查询却没有查询到 Client2 插入的值，这表明避免了幻象现象。

继续执行表 4-7 的操作，如下所示。

表 4-7 隔离级别为“SERIALIZABLE”的并发表 2

Client1	Client2
BEGIN;	
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row)	
	INSERT INTO bluesea values (2,'Pg-02',2); 执行成功，插入操作没有受到 Client1 的影响
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row)	
隔离级别为“SERIALIZABLE”，不存在幻象	ROLLBACK;

然后在 Client2 中执行如下查询锁情况的 SQL，得到这两个并发事务的加锁情况如下。其中，锁“RowExclusiveLock”是 Client2 是施加的，其余的锁是 Client1 施加的。

```
postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
bluesea      | relation |          | 12401 | 57344 |      | 3/5 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation |          | 12401 | 57347 |      | 2/6 | 9164 | AccessShareLock | t |
bluesea      | relation |          | 12401 | 57344 |      | 2/6 | 9164 | AccessShareLock | t |
bluesea      | tuple   |          | 12401 | 57344 | 0 | 1 | 2/6 | 9164 | SIReadLock | t |
bluesea_pkey | page    |          | 12401 | 57347 | 1 |  | 2/6 | 9164 | SIReadLock | t |
(5 rows)
```

这时，Client1 中加锁的情况就和第一次有所不同。Client1 在 relation 上施加了 AccessShareLock 锁，在其索引上也施加了 AccessShareLock 锁，这点和第一次相同；但是，Client1 在 page 和 tuple 上分别施加了谓词读锁“SIReadLock”，而第一次是施加在了 relation 上，这是因为首次执行的时候，表 bluesea 中没有元组，只能施加锁在 relation 上。对于索引，“SIReadLock”只能施加在页面一级，并发度较低。

而并发执行 INSERT 的过程中，不在元组上施加元组锁，而是在 relation 上施加 RowExclusiveLock 锁，如果被插入的表有索引，则索引上会临时施加 RowExclusiveLock 锁，等操作完成即释放而不是等到事务结束时才释放。

对于幻读异常，PostgreSQL 通过 MVCC 和快照技术，在 Client1 中第二次读取数据的时候，判断元组可见性（参见 9.2.1 节的“2. 快照可见性判断”）时，发现元组是新插入没



有提交且是本事务的快照之后发生的事务执行的插入（判定元组对应的事务在快照分为内，不是快照之前的事务插入的可见的数据），判定数据不可见，因此避免了幻读异常。

## 2. 可重复读隔离级别

对于可重复读隔离级别，并发执行如下命令，如表 4-8 所示。

表 4-8 隔离级别为“REPEATABLE READ”的并发表

Client1	Client2
BEGIN;	
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; 输出结果是 c1 的值为 1	
C1: 查锁的状态	
	INSERT INTO bluesea values (2,'Pg-02',2); 插入成功，表明 Client1 的 REPEATABLE READ 不阻塞 Client2 的事务
	C2: 查锁的状态
SELECT c1 FROM bluesea WHERE c1<10; 输出结果是 c1 的值为 1 再次读，读不到 Client2 插入的数据， 不存在幻象	
	UPDATE bluesea SET c1=10 WHERE c1=1; 命令执行成功，没有被阻塞
	C3: 查锁的状态
	COMMIT;
SELECT c1 FROM bluesea WHERE c1<10; 输出结果是 c1 的值为 1 再次读，读不到 Client2 提交的数据，因为 PostgreSQL 使用了 多版本和快照的原因，只能看到本事务初始时快照对应的状态	

在上述 SQL 执行的过程中，检查锁的状态，对应表中的检查步骤与结果如下：

“C1：查锁的状态”，SELECT 操作只施加了 AccessShareLock 锁：

```
postgres=# SELECT * FROM active_locks;
relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
bluesea_pkey | relation |  | 12401 | 57347 |  | 2/9 | 9164 | AccessShareLock | t | 
bluesea      | relation |  | 12401 | 57344 |  | 2/9 | 9164 | AccessShareLock | t | 
(2 rows)
```

“C2：查锁的状态”，INSERT 操作施加了 RowExclusiveLock 锁：

```
postgres=# SELECT * FROM active_locks;
relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
```

```

-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
bluesea      | relation | 12401 | 57344 | | | 3/14 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation | 12401 | 57347 | | | 2/9  | 9164 | AccessShareLock  | t |
bluesea      | relation | 12401 | 57344 | | | 2/9  | 9164 | AccessShareLock  | t |
(3 rows)

```

“C3：查锁的状态”，UPDATE 在索引上操作施加了 RowExclusiveLock 锁：

```

postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
bluesea_pkey | relation | 12401 | 57347 | | | 3/14 | 5888 | RowExclusiveLock | t |
bluesea      | relation | 12401 | 57344 | | | 3/14 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation | 12401 | 57347 | | | 2/9  | 9164 | AccessShareLock  | t |
bluesea      | relation | 12401 | 57344 | | | 2/9  | 9164 | AccessShareLock  | t |
(4 rows)

```

### 3. 已提交读隔离级别

对于已提交读隔离级别，并发执行如下命令（先执行“DELETE FROM bluessea;”清理环境），如表 4-9 所示。

表 4-9 隔离级别为“READ COMMITTED”的并发表

Client1	Client2
	INSERT INTO bluesea values (1,'Pg-01',1);
BEGIN;	
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row)	
	INSERT INTO bluesea values (2,'Pg-02',2); 命令执行成功 C1：查锁的状态
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row) 再次读，读不到 Client2 插入的数据，不存在幻读异常	
	UPDATE bluesea SET c1=10 WHERE c1=1;
	C2：查锁的状态

(续)

Client1	Client2
<pre>SELECT c1 FROM bluesea WHERE c1&lt;10; c1 ---- 1 (1 row) 第三次读，读不到 Client2 更新的数据，不存不可重复读异常</pre>	
	COMMIT;
<pre>SELECT * FROM bluesea; c1   c2   c3 ----+-----+----  2   Pg-02   2 10   Pg-01   1 (2 rows) 第四次读，读到 Client2 提交的数据，是因为隔离级别是读已提交，所以 Client2 提交的数据可见。</pre>	

在上述 SQL 执行的过程中，检查锁的状态，对应表中的检查步骤与结果如下：

“C1：查锁的状态”，INSERT 操作施加了 RowExclusiveLock 锁：

```
postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
bluesea      | relation | 12401 | 57344 | | | 3/16 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation | 12401 | 57347 | | | 2/18 | 9164 | AccessShareLock | t |
bluesea      | relation | 12401 | 57344 | | | 2/18 | 9164 | AccessShareLock | t |
(3 rows)
```

“C2：查锁的状态”，UPDATE 操作在索引上施加了 RowExclusiveLock 锁：

```
postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
bluesea_pkey | relation | 12401 | 57347 | | | 3/16 | 5888 | RowExclusiveLock | t |
bluesea      | relation | 12401 | 57344 | | | 3/16 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation | 12401 | 57347 | | | 2/18 | 9164 | AccessShareLock | t |
bluesea      | relation | 12401 | 57344 | | | 2/18 | 9164 | AccessShareLock | t |
(4 rows)
```

4. 未提交读隔离级别

对于未提交读隔离级别，并发执行如下命令（先执行“DELETE FROM bluesea;”清理环境），如表 4-10 所示。





表 4-10 隔离级别为 'READ UNCOMMITTED' 的并发表

Client1	Client2
	INSERT INTO bluesea values (1,'Pg-01',1);
BEGIN;	
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row)	
	INSERT INTO bluesea values (2,'Pg-02',2); 命令执行成功
	C1: 查锁的状态
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row) 再次读，读不到 Client2 插入的数据，不存在幻读异常	
	UPDATE bluesea SET c1=10 WHERE c1=1;
	C2: 查锁的状态
SELECT c1 FROM bluesea WHERE c1<10; c1 ---- 1 (1 row) 第三次读，读不到 Client2 更新的数据，不存在不可重复读异常，未提交的数据更新操作的结果也读不到，不存在脏读异常	
	COMMIT;
SELECT * FROM bluesea; c1   c2   c3 ----+-----+---- 2   Pg-02   2 10   Pg-01   1 (2 rows) 第四次读，读到 Client2 提交的数据，是因为隔离级别是已提交读，所以 Client2 提交的数据可见	

在上述 SQL 执行的过程中，检查锁的状态，对应表中的检查步骤与结果如下：  
“C1：查锁的状态”，INSERT 操作施加了 RowExclusiveLock 锁：

```
postgres=# SELECT * FROM active_locks;  
relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock  
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
```



```

bluesea      | relation | 12401 | 57344 | | | 3/17 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation | 12401 | 57347 | | | 2/21 | 9164 | AccessShareLock  | t |
bluesea      | relation | 12401 | 57344 | | | 2/21 | 9164 | AccessShareLock  | t |
(3 rows)

```

“C2：查锁的状态”，UPDATE 操作在索引上施加了 RowExclusiveLock 锁：

```

postgres=# SELECT * FROM active_locks;
relname | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
bluesea_pkey | relation | 12401 | 57347 | | | 3/17 | 5888 | RowExclusiveLock | t |
bluesea      | relation | 12401 | 57344 | | | 3/17 | 5888 | RowExclusiveLock | t |
bluesea_pkey | relation | 12401 | 57347 | | | 2/21 | 9164 | AccessShareLock  | t |
bluesea      | relation | 12401 | 57344 | | | 2/21 | 9164 | AccessShareLock  | t |
(4 rows)

```

仔细观察已提交读和未提交读，二者的过程一样，结果一样，未提交读中的示例中，当 UPDATE 操作完成后，Client1 中依旧读不到新的更新数据，只有 Client2 执行了提交才能读取到，这是因为 PostgreSQL 虽然在语法上支持未提交读隔离级别，但实际上消除了脏读异常。

### 4.3.2 写偏序异常

PostgreSQL 在可串行化隔离级别消除了写偏序异常（注意，并发的的事务都需要是可串行化隔离级），示例如下<sup>①</sup>。

#### 1. 数据准备

```

CREATE TABLE dots (
    id INT NOT NULL PRIMARY KEY,
    color VARCHAR(20) NOT NULL
);

INSERT INTO dots VALUES (1, 'black');
INSERT INTO dots VALUES (2, 'white');
INSERT INTO dots VALUES (3, 'black');
INSERT INTO dots VALUES (4, 'white');

```

#### 2. 可串行化隔离级别

对于可串行化隔离级别，并发执行如表 4-11 中所示的命令。

表 4-11 隔离级别为“SERIALIZABLE”的并发表

Client1	Client2
BEGIN;	
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	BEGIN;

① 更多示例，参见：<https://wiki.postgresql.org/wiki/SSI>。



(续)

Client1	Client2
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
C1: 查锁的状态	
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新成功, 此更新操作没有被第一个更新阻塞
	C2: 查锁的状态
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
	SELECT COUNT(*) FROM dots WHERE color = 'black'; count ----- 0 (1 row)
	COMMIT;
COMMIT; ERROR: could not serialize access due to read/write dependencies among transactions DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt. HINT: The transaction might succeed if retried. 报告发生了写偏序异常, 事务被回滚	注意: 此示例是 Client2 先提交。如果是 Client1 先提交, 然 后在 Client2 提交的时候, 也会报告如左面序列化错误。 另外, 如上这点表明, PostgreSQL 对于 MVCC 技术, 采取的是“先提交者获胜”原则。但是, 本示例更新的 对象不是同一个元组, 如果我们更新同一个元组, 则第 二个更新语句会被阻塞, 这意味着“先更新者获胜”。 注意以上两个原则的适用情况不同。
	SELECT * FROM dots; id   color ----+----- 2   white 4   white 1   white 3   white (4 rows) 如果不能发现写偏序异常, 则最后查询的数据结果, 不会是这样, 而是类似如下内容:
	postgres=# SELECT * FROM dots; id   color





(续)

Client1	Client2
	<pre> -----+----- 2   black 4   black 1   white 3   white (4 rows) </pre>

在上述 SQL 执行的过程中，检查锁的状态，对应表中的检查步骤与结果如下：

“C1：查锁的状态”，UPDATE 操作施加了 RowExclusiveLock 锁，可串行化隔离级别使得施加了谓词读锁 SIReadLock：

```

postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
dots_pkey | relation | 12401 | 57352 | 57352 | 2/46 | 9164 | 9164 | RowExclusiveLock | t |
dots      | relation | 12401 | 57349 | 57349 | 2/46 | 9164 | 9164 | RowExclusiveLock | t |
dots      | relation | 12401 | 57349 | 57349 | 2/46 | 9164 | 9164 | SIReadLock       | t |
(3 rows)

```

“C2：查锁的状态”，UPDATE 操作在索引上施加了 RowExclusiveLock 锁，可串行化隔离级别使得施加了谓词读锁 SIReadLock，所以出现了两个 SIReadLock：

```

postgres=# SELECT * FROM active_locks;
relnam | locktype | database | relation | page | tuple | virtualtransaction | pid | mode | granted | xid_lock
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
dots_pkey | relation | 12401 | 57352 | 57352 | 3/30 | 5888 | 5888 | RowExclusiveLock | t |
dots      | relation | 12401 | 57349 | 57349 | 3/30 | 5888 | 5888 | RowExclusiveLock | t |
dots_pkey | relation | 12401 | 57352 | 57352 | 2/46 | 9164 | 9164 | RowExclusiveLock | t |
dots      | relation | 12401 | 57349 | 57349 | 2/46 | 9164 | 9164 | RowExclusiveLock | t |
dots      | relation | 12401 | 57349 | 57349 | 2/46 | 9164 | 9164 | SIReadLock       | t |
dots      | relation | 12401 | 57349 | 57349 | 3/30 | 5888 | 5888 | SIReadLock       | t |
(6 rows)

```

### 3. 可重复读隔离级别

对于可重复读隔离级别，并发执行如下命令，如表 4-12 所示（数据保持初始状态），不能消除写偏序异常（已提交读、未提交读这两个隔离级别也不能消除写偏序异常）。

表 4-12 隔离级别为“REPEATABLE READ”的并发表

Client1	Client2
BEGIN;	
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	BEGIN;
	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;



(续)

UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新成功, 此更新操作没有被第一个更新阻塞
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
	SELECT COUNT(*) FROM dots WHERE color = 'black'; count ----- 0 (1 row)
COMMIT;	
	COMMIT; 注意: 此示例此时, 没有检测到写偏序异常, 提交成功。
	SELECT * FROM dots; id   color ----+----- 2   black 4   black 1   white 3   white (4 rows) 观察这个结果, 如果串行执行, 如先全部执行 Client1 中的命令, 然后执行 Client2 中的命令, 则最终不会有如上结果。Client1 先 Client2 后, 则全部是 white; Client2 先 Client1 后, 则全部是 black

## 4.4 本章小结

本章从三个角度, 简略讲述了 PostgreSQL 的事务管理方式、封锁的相关内容, 以及不同隔离级别与数据异常现象的关系, 为深入剖析 PostgreSQL 的事务管理技术和并发控制技术打下初步的基础。

在第 7、8、9 章, 我们将详细讲述 PostgreSQL 的事务管理技术和并发控制技术, 并剖析源码, 深入探索实现原理与实现方式。



## InnoDB 事务管理与并发控制

InnoDB 支持 ACID，对于不同的特性，分别使用了如下技术予以支持：

- A：原子性，通过提供事务的完整管理模型，实现事务的提交和回滚功能（回滚段），事务的自动提交功能，以支持原子性。
- C：一致性，通过提供“doublewrite buffer”和“crash recovery”功能，以支持数据在运行期间的一致（事务故障）和故障发生后的一致（用日志处理系统故障、介质故障）。另外，通过锁和 MVCC 机制来保证运行期间数据被并发修改情况下的一致。
- I：隔离性，通过“SET ISOLATION LEVEL”语句来设置满足 ANSI SQL 标准规定的四种隔离级别，实现并发事务的不同场景的隔离。通过 MVCC 的快照隔离实现并发事务对同一个数据项的读写隔离。
- D：持久性，通过预写日志（WAL）、备份恢复、双写缓存（doublewrite buffer）等技术实现了数据的持久化存储。

下面，我们将对事务管理和并发控制技术进行讨论，这些技术涉及了 A、C、I 这几个特性。

### 5.1 InnoDB的事务模型

MySQL 使用 InnoDB 作为自己的默认存储引擎，事务管理是由 MySQL Server 实现框架和接口定义，由 InnoDB 提供具体的事务操作和并发控制，所以 MySQL 的事务模型，主要是指 MySQL 的 InnoDB 的事务管理部分。MySQL 是一个处于上层的 Server，提供了





SQL 解析、查询优化、SQL 执行等功能，事务管理和存储交由下层以插件方式提供。不同的插件提供的基本功能不同，如 InnoDB 提供了事务管理的功能，MyISAM 则没有事务管理的功能。

本节讲述 MySQL 中 InnoDB 的事务管理功能和实现。

### 5.1.1 开始事务

在 MySQL 中，使用如下的 SQL 语句，完成对事务的管理：

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]
transaction_characteristic:
    WITH CONSISTENT SNAPSHOT
    | READ WRITE
    | READ ONLY

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1} //事务是否自动启动，默认情况是自动提交
```

MySQL 允许事务自动启动或显式使用“START/BEGIN”来启动。显式使用“START/BEGIN”启动的事务需要使用“COMMIT”或“ROLLBACK”来提交或回滚。

在事务开始的时候，可以指定为“READ WRITE”或“READ ONLY”为事务的访问模式。如果是一个“READ ONLY”的事务，执行的全程是不必加锁的，加锁会降低并发事务执行的并发度。但需要注意的是，在 MySQL 中，“READ ONLY”对于临时表的使用是需要加锁的，如查询语句中带有 GROUP-BY 或 ORDER-BY 等语句导致的排序操作会触发使用临时表。之所以能够提供只读事务这样的功能，全是依赖于 MVCC 机制，MVCC 使得基于锁的并发控制技术的封锁范围大幅减少，读写操作互不阻塞，这样基于锁的事务管理器能够区分只读事务和读写事务，从而实现只读事务全程不加锁的功能。MySQL、PostgreSQL 和其他使用了 MVCC 技术的数据库都有能力把事务区分为“READ WRITE”和“READ ONLY”两种。而单纯基于封锁技术的 Informix 就不能提供只读事务这样的功能。

“WITH CONSISTENT SNAPSHOT”参数表示查询操作要做一个“consistent read”，或称为“一致性无锁读”。“一致性无锁读”受隔离级别的影响，在“REPEATABLE READ”和“READ COMMITTED”级别下获取读数据依赖的快照时间点不同。具体详情，参见 5.3.2 节。

“AND CHAIN”表示当前事务结束后开始一个新事务，这个新事务继承当前事务的隔离级别。

InnoDB 的事务管理器，支持多层嵌套的子事务间错误处理，即在一个事务中，每个语句都是一个“子事务块”，每个子事务块的上下文执行环境独立，各自互不影响。当一个子



句出现错误，整个事务中之前正确执行过的子句不受累回滚。例如：

```
CREATE TABLE bluesea (c1 TINYINT UNSIGNED, c2 VARCHAR(20), c3 TINYINT UNSIGNED,
PRIMARY KEY(c1)); //c1列有主键，默认是InnoDB引擎
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO bluesea values (1,'InnoDB-01',1); //第一个执行成功
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO bluesea VALUES (1,'InnoDB-01',1); //第二个主键冲突执行失败
ERROR 1062 (23000):
mysql> COMMIT; //事务可以成功提交
Query OK, 0 rows affected (0.03 sec)
mysql> SELECT * FROM bluesea; //第一个执行成功的插入语句的结果成功提交，
//不受第二个失败操作的影响

+----+-----+-----+
| c1 | c2          | c3 |
+----+-----+-----+
| 1 | InnoDB-01 | 1 |
+----+-----+-----+
1 row in set (0.00 sec)
mysql> BEGIN; //重新开始事务
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO bluesea VALUES (2,'InnoDB-02',2); //插入新值
Query OK, 1 row affected (0.02 sec)
mysql> CREATE TABLE ttt(A INT); //创建新表，此DDL语句会把上一句INSERT语句的动作提交
Query OK, 0 rows affected (0.28 sec)
mysql> ROLLBACK; //1 不回滚INSERT语句，因为已经被上一DDL语句提交；2 不回滚DDL语句，
//因为DDL语句单独成事务
Query OK, 0 rows affected (0.00 sec)
```

### 5.1.2 提交事务与回滚事务

事务的提交有三种模式：一是自动提交；二是显式执行 COMMIT 提交；三是因执行 DDL 等操作触发之前的 SQL 被隐式提交。

如果不使用 BEGIN 语句主动开启一个事务块，MySQL 会隐式地自动开启一个事务块，单个 SQL 语句为一个事务，执行正确则自动提交，执行失败则自动回滚。

而 DDL 语句，如果处于一个事务块内部，会主动先提交事务 DDL 语句之前执行的语句，并隐式开启一个新的事务。这点和 PostgreSQL 不同。示例如下：

```
CREATE TABLE bluesea (c1 TINYINT UNSIGNED, c2 VARCHAR(20), c3 TINYINT UNSIGNED,
PRIMARY KEY(c1)); //c1列有主键，默认是InnoDB引擎
mysql> INSERT INTO bluesea values (1,'InnoDB-01',1); //第一个执行成功

mysql> BEGIN; //开启一个事务
Query OK, 0 rows affected (0.00 sec)
```





```
mysql> INSERT INTO bluesea values (2,'InnoDB-02',2); //第二个执行成功
mysql> CREATE TABLE test1 (x INT); //在事务块内, 执行DDL操作
Query OK, 0 rows affected (0.20 sec)
mysql> ROLLBACK; //回滚事务。问题: 插入语句和建表语句应该被回滚掉吧?
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM bluesea; //事务块内执行的插入语句, 被提交----被建表语句隐式提交
+-----+-----+
| c1 | c2          | c3 |
+-----+-----+
| 1 | InnoDB-01 | 1 |
| 2 | InnoDB-02 | 2 |
+-----+-----+
2 rows in set (0.00 sec)
mysql> SELECT * FROM test1; //事务块内执行的建表语句, 被隐式提交, 回滚操作对于DDL
语句无效, 所以test1表存在
Empty set (0.01 sec)
```

在基于封锁的并发控制技术中, 常规的实现方式, 都是 2PC。对于 InnoDB 而言, InnoDB 的事务管理器也遵循 2PC。事务的提交标志, 是在内存中设置了 “TRX\_STATE\_COMMITTED\_IN\_MEMORY” 标志 (lock\_trx\_release\_locks() 函数), 然后才开始释放锁等资源, 源码分析部分的 10.3.3 节, 我们将详细讨论。

事务的回滚有三种模式, 一是执行 ROLLBACK 命令主动回滚, 二是因参数 “innodb\_lock\_wait\_timeout” 表示锁等待发生多久事务中的语句进行语句级回滚, 三是因参数 “innodb\_rollback\_on\_timeout” 表示锁等待发生多久后整个事务回滚。

### 5.1.3 MySQL 的 XA

MySQL 的 XA 事务管理基于《X/Open CAE document Distributed Transaction Processing: The XA Specification》规范实现。

对于 MySQL 而言, 在遵守 XA 规范的前提下, 可以视客户端为事务管理器, MySQL Server 为资源管理器。

MySQL 的 XA 实现, 使用 2PC (two-phase commit, 两阶段提交), 但是, 在某些特定情况下, 事务管理器可以使用 1PC (one-phase commit, 一阶段提交)。例如, 事务管理器发现只有一个分支节点, 即全局事务中只有一个资源管理器, 则可以直接告诉资源管理器同时执行 PREPARE 和 COMMIT (此种情况下, 事务是否成功其实是由资源管理器决定的)。

在 MySQL 的一个客户端使用 XA 的基本方式如下:

```
mysql> XA START 'xatest'; //客户端发起一个XA事务
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO mytable (i) VALUES(10); //事务的内容
Query OK, 1 row affected (0.04 sec)
mysql> XA END 'xatest'; //标识事务完毕
```





```
Query OK, 0 rows affected (0.00 sec)
mysql> XA PREPARE 'xatest'; //准备进行2PC中的第一阶段
Query OK, 0 rows affected (0.00 sec)
mysql> XA COMMIT 'xatest'; //进行2PC中的第二阶段
Query OK, 0 rows affected (0.00 sec)
```

## 5.2 InnoDB基于锁的并发控制

InnoDB 使用锁和 MVCC 技术实现了并发事务的访问并发控制。其中，锁是并发控制的基础，在此基础上，实现了 MVCC 机制，用以提高基于锁的方式带来的低效问题，使得读-写、写-读两种操作互不阻塞，提高了单纯基于锁技术的并发效率。

### 5.2.1 基于封锁技术实现基本的并发控制

#### 1. 锁的粒度

InnoDB 提供四种粒度的锁，分别是：

- S：共享锁，或称为读锁。
- X：排他锁，或称为写锁。
- IS：意向共享锁，或称为意向读锁。
- IX：意向排他锁，或称为意向写锁。

这四种锁的相容性，如表 5-1 所示。表中 N 表示不相容，Y 表示相容，新申请的锁可以升级。

例如：“SELECT ... LOCK IN SHARE MODE”语句能够申请意向读锁；“SELECT ... FOR UPDATE”语句能够申请意向写锁。相对于 Informix 的锁粒度，InnoDB 的锁粒度乍看起来还是有些粗糙的（对比表 3-2），但是，MySQL 和 InnoDB 合起来，实现了元数据锁（MDL，参看 11.4.1 节）和记录锁（参看 11.3 节），把锁分的更为细致。其中，元数据锁没有在本章进行解释，相关内容参见 11.4 节。

表 5-1 InnoDB 锁相容性矩阵表

		已经施加的锁			
		X	IX	S	IS
准备申请的锁	X	N	N	N	N
	IX	N	Y	N	Y
	S	N	N	Y	Y
	IS	N	Y	Y	Y

#### 2. 封锁机制

InnoDB 的封锁机制，基本上符合 SS2PL 协议，加锁和解锁分为两个阶段，事务的结束



(提交或回滚)在解锁前完成。

每种粒度的锁的施加时机和持锁时间,和隔离级别及MVCC机制紧密相关。在“SERIALIZABLE”隔离级别下,InnoDB严格遵守SS2PL协议,此时意味着MVCC机制不会发生作用。如果是其他隔离级别,MVCC机制则使得基于锁的并发控制协议所禁止的读-写、写-读并发操作得到解禁,并发效率得到提高。基于锁的并发控制原理在第2章曾经详述过,请参阅。

SELECT操作在“REPEATABLE READ”和“READ COMMITTED”级别下,使用的是基于MVCC机制的快照做一致性无锁读,所以读数据不加锁。只有“SERIALIZABLE”级别,读数才加锁但也在使用MVCC技术获取一个可串行化的快照并判断元组的可见性。

## 5.2.2 锁的种类

MySQL支持多种存储引擎,不同的存储引擎有着不同的封锁机制。MyISAM和MEMORY存储引擎使用表级锁(table-level locking);BDB存储引擎使用页面锁(page-level locking),也支持表级锁;InnoDB存储引擎支持行级锁(row-level locking),MySQL Server支持表级锁,从MySQL Server层面往事务和存储引擎InnoDB看,整个系统默认使用行级锁。

下面我们讨论InnoDB的锁的种类。

为了实现基于锁的并发控制,InnoDB实现了多种类型的锁,包括在索引组织表上实现行级的记录锁(Record lock)、间隙锁(gap)和元组所与间隙锁组合而成的范围类型的锁“Next-key”、类似“gap”锁的插入意向锁(Insert intention lock),以及支持空间索引的谓词锁(Predicate Lock)。这些锁的实现,既依赖于封锁原理又依赖于InnoDB特定的表组织结构,然后实现不同的封锁作用。

此外,InnoDB支持一种称为自增锁(AUTO-INC Lock)的锁,自增锁类似Oracle和PostgreSQL中的序列对象,也类似SQL Server中的自增列。严格地讲,这种锁和并发访问控制用于互斥并发事务操作的锁不属于同一类型,类似系统级的共享资源上施加的对象互斥锁,因此我们不做深入探讨。

下面我们详细讲述这些不同类型的锁。

### 1. 记录锁(Record Locks)

InnoDB的锁的粒度是行级锁,这样的行级锁在InnoDB中称为记录锁“record lock”(实则是索引上的记录之锁)。

因为InnoDB的表的组织结构,是一个索引组织表(Oracle支持堆表,也支持索引组织表;PostgreSQL只支持堆表,不支持索引组织表;InnoDB只支持索引组织表),所以在真实的元组之上存在一棵构建在主键上的B+树,这棵B+树的树根节点是获取数据的入口。

因为有这样的B+树索引存在,使得InnoDB对数据的扫描方式(MySQL的优化器支持全表扫描和索引扫描的概念,但全表扫描的实现方式,还是先在B+树上执行,然后通过叶

子节点间的指针完成顺序扫描,这种扫描方式施加的是表级锁),首先是在 B+ 树上进行的,所以 InnoDB 提供的记录锁,实则是在 B+ 树上的索引记录锁。

需要注意的是,InnoDB 的记录锁是针对索引加锁,不是针对物理记录加锁,所以虽然是访问不同行的记录,但是如果是使用相同的索引键,将出现锁冲突。从这点看,InnoDB 的记录锁锁定的范围大于其他数据库如 PostgreSQL 行锁锁定的范围,粒度有些粗,这意味着并发度低。

## 2. 间隙锁 (Gap Locks)

两个索引项(索引记录)之间的间隔,称为间隙,把这个间隙视为一个对象,在此对象上加锁,就是间隙锁(gap lock)。第一个索引项之前和最后一个索引项之后的间隙,也可以施加间隙锁。

间隙锁和共享锁、排它锁等锁的粒度的分类角度是不同的,前者基于锁要施加的对象,后者从操作的类型角度出发表达加锁的操作类型。所以实践中可以看到,存在有“共享间隙锁”“排他间隙锁”等说法是正确的。

在 InnoDB 中,间隙锁有时存在需要合并的可能。例如,事务 T1 因执行查询语句在索引项值为“8”之前的间隙上持有一个共享间隙锁,事务 T2 因执行清理(purge 操作)打算在同一个间隙上持有排他间隙锁,按照 InnoDB 锁的相容性表 5-1 看,共享锁排斥排它锁获取,但是因为加锁对象是间隙,InnoDB 则允许这两种锁合并。

间隙锁的主要作用,就是和记录锁组合成为 Next-Key 锁,解决幻象异常现象。

间隙锁在不同的隔离级别下,存在有不同的作用范围。能发挥间隙锁作用的,是“REPEATABLE READ”隔离级别,在这个级别下使用带有间隙锁的 Next-Key 锁,解决了幻象的问题(具体解决方式参加下一个标题的内容)。

如果隔离级别是“READ COMMITTED”,则间隙锁的作用仅在如下情况下有效:

- ❑ 外键约束(foreign-key constraint)检查。
- ❑ 重复键(duplicate-key)检查。
- ❑ 半一致(semi-consistent)读<sup>①</sup>:执行 UPDATE 语句的时候,InnoDB 返回处于已经提交状态的最新的元组给 MySQL Server,由 MySQL Server 决定得到的元组是否满足 UPDATE 的 WHERE 条件。使用半一致读时间隙锁被边使用边释放。

间隙锁在隔离级别是“READ COMMITTED”时,不能在搜索(searches)和索引扫描(index scans)时起作用。

## 3. Next-Key Locks

Next-Key 锁,由记录锁和此记录前的间隙上的间隙锁组成。

<sup>①</sup> 半一致读是“READ COMMITTED”与consistent read合作的结果。其生效的条件是:是READ COMMITTED隔离级别;或者是REPEATABLE READ隔离级别,且设置了innodb\_locks\_unsafe\_for\_binlog参数为TRUE值。



下面我们举例说明 Next-Key 锁在“REPEATABLE READ”隔离级别下是如何解决幻象问题的。

假定 InnoDB 的索引包括 10、18、53 和 80，在“REPEATABLE READ”隔离级别下，执行查询时，因为 Next-Key 锁存在，则 Next-Key 锁的锁定范围如下：

- $(-\infty, 10]$ ：锁定索引项 10 和 10 之前的间隙，因为 10 之前没有其他索引项，所以为负无穷。
- $(10, 18]$ ：锁定索引项 11，同时锁定 10 和 18 之间的间隙。不包括 10，包括 18。
- $(18, 53]$ ：同上。
- $(53, 80]$ ：同上。
- $(80, \infty)$ ：锁定索引项 80 和 80 之后的间隙，因为 20 之后没有其他索引项，所以为正无穷。

假定事务 T1 查询条件为“WHERE key=10”，事务 T2 执行 NSERT 操作，条件也是“WHERE key=10”，因为事务 T1 施加了 Next-Key 锁在  $(-\infty, 10]$  上，导致事务 T2 不能获取锁处于等待状态，这样就不能发生幻象异常。

如果隔离级别是“READ COMMITTED”，因为间隙锁不能起作用，对于上述索引事例和 WHERE 条件，索引项 10 上因不能有间隙锁，则锁的范围就是  $[10, 10]$ ，即只能锁定已经存在的索引项 10 这个对象，新插入的对象 10 不可能被事务 T1 锁定，所以事务 T2 的插入操作能够成功，导致幻象异常发生。

#### 4. Insert Intention Locks

Insert Intention 锁，基于间隙锁，专门用于 INSERT 操作。在一个 Next-Key 锁锁的范围内，对于不产生冲突的 INSERT 操作，尽管存在间隙锁排斥其锁定的范围内进行的其他操作，但是此处开放一个小特权，就是允许不产生冲突的插入操作得到执行。这样能够提高插入操作的并发度。

例如，假定事务 T3 插入 13，事务 T4 插入 15，事务 T5 插入 16，这三个事务在如前面的索引上，插入数据的范围在  $(10, 18]$  内，且插入的值分别是 13、15、16，互不冲突，则 Insert Intention 锁表明这三个事务的插入操作是被允许的。这就是 Insert Intention 锁。

#### 5. Predicate Locks for Spatial Indexes

InnoDB 支持空间索引，如果使用 Next-Key 锁来支持空间索引，则不能胜任，这是因为普通的索引都是键值类型，意味着索引存在一个方向，这个方向是单向的，要么是升序方向要么是降序方向，这个方向的存在，使得数据库引擎可以利用索引进行常规的范围查询。

但是，在空间数据类型面前，这个单向的有序变得失去了作用，因为空间数据是多维多向的，是以区域或空间为范围的，没有确定的方向顺序，所以单向的 Next-Key 锁便不再能使用。

空间索引是建立在“Minimum Bounding Rectangle (MBR)”上的，InnoDB 为索引项上的 MBR 增加了一个谓词锁，实现空间索引上的并发控制。

### 5.2.3 锁的施加规则

MySQL 支持各种 SQL 语句，从事务的角度看，对于 SQL 语句的认识，应该从并发控制技术的角度来进行。MySQL 的 InnoDB 对于 SQL 语句提供了事务操作的支持，而这样的支持是采用并发控制技术中的封锁技术完成的，所以在此我们基于事务锁来讨论锁的施加规则。

SQL 语句可以分为数据定义语言 (DDL)、数据控制语言 (DCL)、数据查询语言 (DQL)、数据操纵语言 (DML) 四种类型的语句，前两种语句，涉及的对象在数据之上，所以加锁的范围，通常是表级，对应表级锁。后两种语句操作的对象是数据，加锁的范围，通常是数据级，这就对应行级锁。但是，也有例外，如 DDL 语句中的 ALTER TABLE 语句包括丰富的子句，不同的子句操作的对象级别是不同的，如重命名表对象操作的是表，而修改列的数据类型，操作的即包括表的元数据又包括表的数据对象。

下面对于 SQL 语句的锁的施加规则的分析，则包括了表级或行级加锁的行为。

SELECT ... FOR UPDATE 或 SELECT ... LOCK IN SHARE MODE：首先，对扫描过的行加锁（索引的记录上施加锁），如果扫描过的行不满足 WHERE 条件，则释放锁。但有的时候，锁的释放并不是很及时，例如 UNION 操作下被扫描过的行可能被放到临时表中，这时锁不会释放，只有在查询结束后才被释放。

- ❑ ALTER TABLE ... LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}：在指定的表上施加读锁或排它锁<sup>①</sup>。
- ❑ CHECKSUM TABLE：为指定的表施加读锁。读取数据符合一致性读。类似的还有 ANALYZE/CHECK TABLE/OPTIMIZE TABLE/REPAIR TABLE 等操作。
- ❑ CREATE TABLE ... SELECT ...：其中的 SELECT 操作符合 SELECT 语句的加锁规则；只是不能带有 FOR UPDATE 子句。
- ❑ DELETE FROM ... WHERE ...：在索引项上施加排他 Next-Key 锁。
- ❑ HANDLER tbl\_name OPEN/READ：直接访问 MYISAM/InnoDB 表时，不加锁，不能确保数据的一致性。
- ❑ INSERT：在被插入的索引项上施加记录锁（也叫插入意向锁）。
- ❑ INSERT ... ON DUPLICATE KEY UPDATE：在被插入的索引项上施加排他 Next-Key 锁。
- ❑ INSERT INTO T SELECT ... FROM S WHERE ...：对于被插入到表 T 中的元组，在其对应的索引项上施加排他记录锁。如果隔离级别是“READ COMMITTED”则在表 S 对应的索引项上不加锁，这是一个一致性读操作；否则，施加共享 Next-Key 锁。
- ❑ FLUSH TABLES WITH READ LOCK：关闭了所有的表之后，对所有的表施加一个全局的读锁<sup>②</sup>。
- ❑ FLUSH TABLES tbl\_name [, tbl\_name] ... WITH READ LOCK：在指定的表对象上

① 参见：<https://dev.mysql.com/doc/refman/5.7/en/alter-table.html#alter-table-concurrency>。

② 源自：<https://dev.mysql.com/doc/refman/5.7/en/flush.html>

Closes all open tables and locks all tables for all databases with a global read lock。

获取表级的读锁。但是加锁的过程存在不同的锁切换的过程，详细参见 MySQL 官方手册<sup>①</sup>。但是，本条命令和上条命令在施加锁的时候，还需要根据隔离级别进行区别，详情参见 11.5.3.3 节对于 `ha_innodb::store_lock()` 函数的分析。

- ❑ `LOCK TABLES t1,...,tn [READ [LOCAL] | [LOW_PRIORITY] WRITE]`：带有“READ”子句是施加表级读锁；带有“READ LOCAL”子句是在施加表级读锁的时候，还允许并发插入操作，但是对于 InnoDB 的表本条不适用；带有“WRITE”子句是施加表级写锁，排斥其他 SESSION 并发访问被加锁的表；带有“LOW\_PRIORITY WRITE”子句只是兼容早期的版本，不再有实际效果<sup>②</sup>。
- ❑ `REPLACE INTO t SELECT ... FROM s WHERE ... 或 UPDATE t ... WHERE col IN (SELECT ... FROM s ...)`：SELECT 语句作为子查询出现在其他子句中，则对标 s 中的数据对应的索引项施加共享 Next-Key 锁。
- ❑ `REPLACE`：如果不会在唯一键上发生冲突，则 REPLACE 执行时施加的锁和 INSERT 相同。否则，冲突发生，则对要被替换的对象对应的索引项施加排他 Next-Key 锁。
- ❑ `SELECT ... LOCK IN SHARE MODE`：在索引项上施加共享 Next-Key 锁。
- ❑ `SELECT ... FROM ... FOR UPDATE`：在索引项上施加排他 Next-Key 锁。这样的锁阻塞 SELECT ... LOCK IN SHARE MODE 操作但不阻塞 SELECT ... FROM 这样的一致性读操作。
- ❑ `SELECT ... FROM` 通常作为一个一致性读操作发生，是不需要施加任何锁的。但是，在隔离级别为“SERIALIZABLE”时需要在索引项上施加相应的共享的 Next-Key 锁。
- ❑ `UPDATE ... WHERE ...`：在索引项上施加排他 Next-Key 锁。

如果一个表上定义了外键约束，那么在触发约束条件被检查的元组所对应的索引项上，无论是执行 INSERT、UPDATE、DELETE 中的哪种操作，无论约束检查是否成功，都会被施加共享 Next-Key 锁。

LOCK TABLES 会在 MySQL Server 层设置表级锁，InnoDB 此时不会加锁。对于 InnoDB 而言，参数设置为“`innodb_table_locks = 1`”，InnoDB 会知道 MySQL Server 层设置了表级锁，否则不知。如果 InnoDB 不知道 MySQL Server 层设置了表级锁，则极容易发生死锁，这是因为 MySQL Server 层和 InnoDB 各自不知道对方的加锁解锁情况，死锁检测机制是无法进行检测的，实践中需要特别注意。

① 源自：<https://dev.mysql.com/doc/refman/5.7/en/flush.html>

This statement flushes and acquires read locks for the named tables. The statement first acquires exclusive metadata locks for the tables, so it waits for transactions that have those tables open to complete. Then the statement flushes the tables from the table cache, reopens the tables, acquires table locks (like LOCK TABLES ... READ), and downgrades the metadata locks from exclusive to shared. After the statement acquires locks and downgrades the metadata locks, other sessions can read but not modify the tables.

② 详细描述参见：<https://dev.mysql.com/doc/refman/5.7/en/lock-tables.html>



### 5.2.4 获取 InnoDB 行锁争用情况

可以通过检查 InnoDB\_row\_lock 状态变量来分析系统上的行锁的争夺情况：

```
mysql> show status like 'innodb_row_lock%';
+-----+
| Variable_name | Value |
+-----+
| InnoDB_row_lock_current_waits | 0 |
| InnoDB_row_lock_time | 0 |
| InnoDB_row_lock_time_avg | 0 |
| InnoDB_row_lock_time_max | 0 |
| InnoDB_row_lock_waits | 0 |
+-----+
5 rows in set (0.01 sec)
```

如果发现锁争用比较严重，如 InnoDB\_row\_lock\_waits 和 InnoDB\_row\_lock\_time\_avg 的值比较高，还可以通过设置 InnoDB Monitors 来进一步观察发生锁冲突的表、数据行等，并分析锁争用的原因。

具体方法如下：

```
mysql> CREATE TABLE innodb_monitor(a INT) ENGINE=INNODB;
//任何库下都可以，放在特定的库内更好
Query OK, 0 rows affected (0.09 sec)
mysql> CREATE TABLE innodb_tablespace_monitor(a INT) ENGINE=INNODB;
Query OK, 0 rows affected (0.12 sec)
mysql> CREATE TABLE innodb_lock_monitor(a INT) ENGINE=INNODB;
Query OK, 0 rows affected (0.13 sec)
mysql> CREATE TABLE innodb_table_monitor(a INT) ENGINE=INNODB;
Query OK, 0 rows affected (0.09 sec)
创建表后 INNODB 会每隔 15 秒输出一 次 INNODB 状态信息到 error log，通过删除表停止该 monitor 功能。
```

另外，在 5.6.16 版本之后，可以通过如下命令进行锁状态的检查。

```
SET GLOBAL innodb_status_output=ON;
SET GLOBAL innodb_status_output_locks=ON;
SHOW ENGINE INNODB STATUS;
```

### 5.2.5 死锁

InnoDB 的并发写操作会触发死锁，InnoDB 也提供了死锁检查的机制。

多数情况下，可以通过 SHOW ENGINE INNODB STATUS 命令查阅死锁的相关信息，这个命令能够帮助定位大部分的死锁问题。

通过设置 innodb\_deadlock\_detect 参数可以打开或关闭死锁检测。如果死锁检测被关闭，设置 innodb\_lock\_wait\_timeout 可以在超时发生后回滚被阻塞的事务。

参数 `innodb_print_all_deadlocks` 被设置为 ON 可以把死锁相关信息保存到 MySQL 错误日志中。

MySQL 中 `INFORMATION_SCHEMA` 包括有 InnoDB 事务锁相关的三个表：`INNODB_TRX` 表、`INNODB_LOCKS` 表、`INNODB_LOCK_WAITS` 表。通过查看这三个表可以掌握事务加锁的情况和锁等待的情况，帮助分析死锁。

11.3.2 节和 11.4.3 节分别详细探讨了记录锁和元数据锁的死锁检测的实现技术，请参阅相关章节。

## 5.3 InnoDB基于MVCC的并发控制

InnoDB 在基于锁的并发控制技术基础上，实现了 MVCC 技术。InnoDB 的 MVCC 技术，有这样几个特点：

- ❑ 事务的标识，依靠事务 ID，是一个全局唯一的 64bits 数值。
- ❑ 多版本，是元组级的多版本，而不像 Oracle 实现的是页面级的多版本。
- ❑ 最新的数据存储在数据页面中，其他数据的旧版本存储在回滚段中。

因为 InnoDB 的多版本是元组级的版本，所以在每个记录上，有一些与并发和回滚等与事务相关的隐含字段<sup>①</sup>，分别为：

- ❑ `DB_TRX_ID`：6 字节长，表示上一个执行插入或更新操作的事务。
- ❑ `DB_ROLL_PTR`：7 字节长，表示旧版本的数据位于回滚段中的位置，指向的是一个旧版本。只有元组被更新，才会有新版本产生，旧版本被置于回滚段。因此一致性无锁读操作按照“read view”快照需要读取旧版本时，只能根据事务 ID 到回滚段中寻找旧版本。
- ❑ `DB_ROW_ID`：6 字节长，表示执行插入操作后生成的单调自增长的行的 ID 标识，如果存在聚集索引，索引项则包括的是这个 `DB_ROW_ID` 值。
- ❑ `DELETE_BIT`：删除标志位。

位于回滚段中的 UNDO 日志分为两种：

- ❑ `INSERT UNDO logs`：插入到回滚段的日志，仅用于事务提交时使用，当事务提交，则插入 UNDO 日志里的内容被清除。
- ❑ `UPDATE UNDO logs`：被用于一致性无锁读，为一致性读提供快照隔离下的可被读取的老版本数据。当没有需要满足一致性读的快照时，一些老版本数据才被清理。

InnoDB 依靠 MVCC 解决封锁技术禁止的写-读、读-写并发操作，提高了并发事务的并发程度。更多详细深入的探讨，请参见第 12 章。

---

① 隐含字段为：`DB_ROW_ID`、`DB_TRX_ID`、`DB_ROLL_PTR`和 `DB_MIX_ID`。

## 5.4 隔离级别与数据异常

### 5.4.1 SQL 标准定义的三种读异常

InnoDB 支持 ANSI SQL 标准定义的四种隔离级别，分别为：可串行化 `SERIALIZABLE`、可重复读 `REPEATABLE READ`、已提交读 `READ COMMITTED` 和未提交读 `READ UNCOMMITTED`。其中，`REPEATABLE READ` 是默认值。设置隔离级别通过“`SET ISOLATION LEVEL`”语句来完成。

这四种隔离级别，分别用于解决三种读异常现象。符合 2.2.1 节我们讨论的基于锁的封锁并发控制技术。但是，因为 InnoDB 使用了 MVCC 并发控制技术，使得 MySQL 在不同隔离级别下的表现与基于锁的 Informix 不完全相同，具体情况可以参见表 5-2、表 5-3、表 5-4 和表 5-5，分别验证了四种隔离级别的多事务并发情况。

首先，构造环境如下：

```
SELECT version(); //以5.7.11版本为例
SELECT @@global.tx_isolation,@@tx_isolation;    //每个客户端的隔离级别默认如下
+-----+-----+
| @@global.tx_isolation | @@tx_isolation |
+-----+-----+
| REPEATABLE-READ      | REPEATABLE-READ |
+-----+-----+
SET AUTOCOMMIT=OFF;
innodb_lock_wait_timeout = 500 //my.ini文件设置
```

#### 1. 可串行化隔离级别

其次，分别启动两个客户端，Client1 和 Client2，假定 Client1 为主事务端，在此端改变隔离级别，执行如表 5-2 中所示的命令：

表 5-2 隔离级别为“`SERIALIZABLE`”表

Client1	Client2
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;	
BEGIN;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; Empty set (0.02 sec)	
	INSERT INTO bluesea values (1,'InnoDB-01',1); 执行后处于等待状态,表示 Client1 的读锁互斥了插入操作的写锁
SELECT c1 FROM bluesea WHERE c1<10; Empty set (0.02 sec)	
隔离级别为 'SERIALIZABLE', 不存在幻象	



当 Client2 处于等待状态时，启动新的客户端 Client3 执行：

```
mysql> SELECT lock_id, lock_trx_id, lock_mode, lock_type, lock_table, lock_index FROM
INFORMATION_SCHEMA.INNODB_LOCKS;
+-----+-----+-----+-----+-----+-----+
| lock_id | lock_trx_id | lock_mode | lock_type | lock_table | lock_index |
+-----+-----+-----+-----+-----+-----+
| 34564:53:3:1 | 34564 | X | RECORD | `d`.`blueseas` | PRIMARY |
| 281475164800816:53:3:1 | 281475164800816 | S | RECORD | `d`.`blueseas` | PRIMARY |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

可以看出，Client1 上的 S 锁互斥了 Client2 想要施加的 X 锁，致使 Client2 处于等待状态，这样就避免了幻象现象。

2. 可重复读隔离级别

对于可重复读隔离级别，并发执行如下命令，如表 5-3 所示。

表 5-3 隔离级别为 “REPEATABLE READ” 表

Client1	Client2
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
INSERT INTO bluesea values (1,'InnoDB-01',1);	
BEGIN;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; 输出结果是 c1 的值为 1	
	INSERT INTO bluesea values (2,'InnoDB-02',2); 插入成功，Client1 的 REPEATABLE READ 不阻塞 Client2 的事务，表明 Client1 的 SELECT 操作在谓词指定的范围内没有施加读锁和间隙锁
SELECT c1 FROM bluesea WHERE c1<10; 输出结果是 c1 的值为 1 再次读，读不到 Client2 插入的数据，不存在幻象	
	UPDATE bluesea SET c1=10 WHERE c1=1; 命令执行成功，没有被阻塞
	COMMIT;
SELECT c1 FROM bluesea WHERE c1<10; 输出结果是 c1 的值为 2 再次读，读不到 Client2 提交的数据，是因为隔离级别是可重复读。另外，读操作不被 INSERT 的写操作阻塞，是因为 InnoDB 使用了快照和 MVCC 的原因	

当 Client2 的 INSERT/UPDATE 命令执行成功时，启动新的客户端 Client3 执行，则没有锁发生等待：

```
mysql> SELECT lock_id, lock_trx_id, lock_mode, lock_type, lock_table, lock_index
FROM INFORMATION_SCHEMA.INNODB_LOCKS;
Empty set (0.00 sec)
```

3. 已提交读隔离级别

对于已提交读级别，并发执行如下命令，如表 5-4 所示。

表 5-4 隔离级别为“READ COMMITTED”表

Client1	Client2
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;	
BEGIN;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; Empty set (0.02 sec)	
	INSERT INTO bluesea values (1,'InnoDB-01',1); 命令执行成功
SELECT c1 FROM bluesea WHERE c1<10; Empty set (0.02 sec) 再次读，读不到 Client2 插入的数据	
	COMMIT;
SELECT c1 FROM bluesea WHERE c1<10; +----+   c1   +----+   1   +----+ 再次读，读到 Client2 提交的数据，是因为隔离级别 是已提交读，所以 Client2 提交的数据可见。	

4. 未提交读隔离级别

对于未提交读级别，并发执行如下命令，如表 5-5 所示。

表 5-5 隔离级别为“READ UNCOMMITTED”

Client1	Client2
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;	
BEGIN;	BEGIN;
SELECT c1 FROM bluesea WHERE c1<10; Empty set (0.02 sec)	
	INSERT INTO bluesea values (1,'InnoDB-01',1); 命令执行成功

(续)

<pre>SELECT c1 FROM bluesea WHERE c1&lt;10; +----+   c1   +----+   1   +----+</pre> <p>再次读，读到 Client2 没有提交的数据，是因为隔离级别是未提交读，所以 Client2 插入的数据可见。</p>	没有执行 COMMIT 或 ROLLBACK，Client1 已经能够读到插入的新数据
--	---

5.4.2 写偏序异常

1. 数据准备

```
CREATE TABLE dots (
    id INT NOT NULL PRIMARY KEY,
    color VARCHAR(20) NOT NULL
);

INSERT INTO dots VALUES (1, 'black');
INSERT INTO dots VALUES (2, 'white');
INSERT INTO dots VALUES (3, 'black');
INSERT INTO dots VALUES (4, 'white');
```

2. 可串行化隔离级别

对于可串行化隔离级别，并发执行如表 5-6 中所示的命令。

表 5-6 隔离级别为“SERIALIZABLE”的并发表

Client1	Client2
BEGIN;	
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;	BEGIN;
	SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
C1: 查锁的状态	
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新被阻塞
C2: 查锁的状态	



(续)

Client1	Client2
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
COMMIT; 成功	
	COMMIT; 成功
	SELECT * FROM dots; +---+-----+   id   color   +---+-----+   1   white     2   white     3   white     4   white   +---+-----+ 4 rows in set (0.00 sec) 观察这个结果，如果串行执行，Client1先Client2后，则全部是white；Client2先Client1后，则全部是black。而此结果表明这是一个可串行化调度，写偏序没有发生

在上述 SQL 执行的过程中，检查锁的状态，对应表中的检查步骤与结果如下：

“C1：查锁的状态”，UPDATE 操作施加了 X 锁，互斥了其他的 UPDATE 操作，但查询不到，这是因为第一个施加的是一个隐式锁，只有在其他锁施加时才会激发隐式锁向显式锁转换：

```
mysql> SELECT lock_id, lock_trx_id, lock_mode, lock_type, lock_table, lock_index
FROM INFORMATION_SCHEMA.INNODB_LOCKS;
Empty set (0.00 sec)
```

“C2：查锁的状态”，两个 UPDATE 操作施加了两个 X 锁，但在索引项上第二个锁被第一个锁阻塞：

```
mysql> SELECT lock_id, lock_trx_id, lock_mode, lock_type, lock_table, lock_index
FROM INFORMATION_SCHEMA.INNODB_LOCKS;
+-----+-----+-----+-----+-----+-----+
| lock_id      | lock_trx_id | lock_mode | lock_type | lock_table | lock_index |
+-----+-----+-----+-----+-----+-----+
| 65575:84:3:2 | 65575      | X        | RECORD   | `d`.`dots` | PRIMARY    |
```

65574:84:3:2   65574	X	RECORD	`d`.`dots`   PRIMARY	
+-----+-----+-----+-----+-----+				
2 rows in set (0.00 sec)				

3. 可重复读隔离级别

对于可重复读隔离级别，并发执行如下命令，如表 5-7（数据保持初始状态）所示，消除写偏序异常。

表 5-7 隔离级别为“REPEATABLE READ”的并发表

Client1	Client2
BEGIN;	
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;	BEGIN;
	SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新被阻塞
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
COMMIT;	
	更新成功完成
	COMMIT;
	注意：
	此示例此时，没有检测到写偏序异常，提交成功。
	SELECT * FROM dots; +---+-----+   id   color   +---+-----+   1   white     2   white     3   white     4   white   +---+-----+ 4 rows in set (0.00 sec) 观察这个结果，如果串行执行，Client1 先 Client2 后，则全部是 white；Client2 先 Client1 后，则全部是 black。而此结果表明这是一个可串行化调度，写偏序没有发生

#### 4. 已提交读隔离级别

对于已提交读隔离级别，并发执行如下命令，如表 5-8（数据保持初始状态）所示，不能消除写偏序异常（已提交读、未提交读这两个隔离级别也不能消除写偏序异常）。

表 5-8 隔离级别为“READ COMMITTED”的并发表

Client1	Client2
BEGIN;	
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;	BEGIN;
	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
C1: 查锁的状态	
	UPDATE dots SET color = 'white' WHERE color = 'black'; 执行成功，更新没有被阻塞
C2: 查锁的状态	
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
COMMIT;	
	更新成功完成
	COMMIT; 注意： 此示例此时，没有检测到写偏序异常，提交成功。
	SELECT * FROM dots; +----+-----+   id   color   +----+-----+   1   white     2   black     3   white     4   black   +----+-----+ 4 rows in set (0.00 sec) 观察这个结果，如果串行执行，Client1 先 Client2 后，则全部是 white；Client2 先 Client1 后，则全部是 black。而此结果表明这不是一个可串行化调度，存在写偏序异常



## 5.5 本章小结

本章首先探讨了事务模型，其次重点探讨了 InnoDB 的并发访问控制技术，包括封锁技术和 MVCC 技术，并探讨了数据异常和隔离级别之间的关系，这些讨论限于 InnoDB 的范围，是对用户表数据的并发操作，所以涉及的是记录锁相关的内容。并且，这些内容在第 11 章和第 12 章详细展开了讨论。

MySQL 因为特殊的插件式结构和历史原因，使得 MySQL Server 层也使用了封锁技术作为并发访问控制的技术，相关内容是元数据锁。这些内容在第 11 章将详细展开了讨论。

## 第 6 章

# Oracle 事务管理与并发控制

Oracle 事务并发控制技术采用典型的 SS2PL 和基于快照的 MVCC 技术，以支持 ACID，但是没有真正实现一致性。对于不同的特性，分别使用了如下技术予以支持：

- ❑ A：原子性，通过提供事务的完整管理模型，在系统的运行过程中实现事务的提交和回滚功能（回滚段），以支持运行原子性。提供 WAL 日志和恢复机制，以支持恢复原子性。
- ❑ C：一致性，通过提供基于封锁技术的 SS2PL，以支持并发操作下元数据的一致；通过 MVCC 技术，以支持并发操作下用户数据的一致；通过 WAL 日志和恢复机制以支持故障发生后的恢复一致（系统故障、介质故障）。
- ❑ I：隔离性，通过“SET ISOLATION...”语句来设置满足 ANSI SQL 标准规定的四种隔离级别中的两种，以快照和多版本构成的 MVCC 技术实现并发事务对同一个数据项的读写隔离，这样实现了一定的并发事务的不同场景的隔离，但没有真正实现可串行化隔离级别。
- ❑ D：持久性，通过预写日志（WAL）、恢复等技术实现了数据的持久化存储。

下面，将对事务管理和并发控制技术进行讨论，这些技术涉及了 A、C、I 这几个特性。事务管理技术与 A 紧密相关，并发控制于 C 和 I 紧密关联。但是，我们不详细描述 Oracle 的并发控制技术和实现细节（Oracle 数据库有着丰富的资料，读者可自行搜索），而是着眼于主要的技术和内容，对比其他数据库，以期读者对事务管理和并发控制技术有更深入的认识。

## 6.1 Oracle 的事务操作

### 6.1.1 事务管理

用户不能主动显式地开启一个事务，而是系统自动地开启一个事务，上个事务结束即

自动开启了下一个事务，这点和主流的其他数据库如 Informix、PostgreSQL、MySQL 等不同。例如：

```
INSERT INTO scott.emp(empno) VALUES(1011);
//没有显式开启事务，作为第一条语句自动开启一个事务
SAVEPOINT AA;                                //定义一个保存点
INSERT INTO scott.emp(empno) VALUES(1012);
ROLLBACK TO AA;                              //回滚到保存点AA
INSERT INTO scott.emp(empno) VALUES(1013);
COMMIT;                                     //必须显式地提交（或回滚）事务。之后，下一个事务已经隐式开启
INSERT INTO scott.emp(empno) VALUES(1015);
ROLLBACK;                                   //必须显式地回滚（或提交）事务。之后，再下一个事务已经隐式开启
```

Oracle 中，当用户想要终止一个事务处理时，必须显示使用 COMMIT 和 ROLLBACK 语句结束。

Oracle 实现了平板事务模型，也提供了保存点功能。定义一个保存点的方式如下：

```
SAVEPOINT savepoint_name
```

在 Oracle 内部，使用保存点来模拟子事务，属于有保存点的平板事务模型，不是真正的嵌套事务模型。保存点可以用如下语句回滚。

```
ROLLBACK TO savepoint_name
```

一个事务，必须使用 COMMIT 和 ROLLBACK 语句结束，这点也和其他数据库不同，也就是说，Oracle 没有在 SERVER 层提供事务自动提交的功能。但是，Oracle 在执行 DDL 语句时（如 Create Table、Create View 等），会在执行之前自动发出一个 Commit 命令，这意味着隐式提交了之前的语句。

Oracle 的事务模型符合 SS2PL，以事务提交所做的工作为例，Oracle 官方文档描述如下：

```
When a transaction commits, the server process performs the following steps:
1. Finds an SCN value
2. Updates the transaction table⊖
3. Puts the current undo block into the free block pool (under some conditions)
4. Creates a commit record in the redo log buffer           //生成提交信息
5. Flushes the redo log buffer to disk (for durability)   //先刷出日志
6. Releases locks held on rows and tables                 //日志刷出后才释放锁--释放行级锁和表级锁
```

## 6.1.2 事务属性和隔离级别

用户可以使用如下命令设置一个事务的属性，只在一个事务块内有效，不影响下一个事务块，如果不想使用属性的默认值，则需要单独为每一个事务块设置属性。

<sup>⊖</sup> The data structure within an undo segment that holds the transaction identifiers of the transactions using the undo segment.



```
SET TRANSACTION READ ONLY
```

```
//事务是只读的，利用MVCC的快照技术建立一个快照，来确保一个事务的读一致点
```

```
SET TRANSACTION READ WRITE
```

//事务是读写混合操作的，但允许全部操作是“读”操作。PostgreSQL遇到全部操作是“读”操作，会自动标识这个事务是只读的，此时有助于SSI技术中快速判断是否存在写偏序异常

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED //设置事务的隔离级别是已提交读
```

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE //设置事务的隔离级别是可串行化，但Oracle没有提供真正的可串行化隔离级别，理论上不能消除所有的数据异常现象

在 Oracle 中，没有 READ UNCOMMITTED 和 REPEATABLE READ 隔离级别，只有 READ COMMITTED 和 SQL 标准一致，确保不会出现脏读。Oracle 没有提供真正的可串行化隔离级别（尽管可以设置隔离级别为“SERIALIZABLE”），理论上不能消除所有的数据异常现象。隔离级别的更多内容，请参见 6.4 节。

### 6.1.3 XA 事务

Oracle 对异构分布式数据库的支持是通过 X/OPEN、XA 实现的。XA 事务的管理，通过库函数提供（类似 Informix），而没有提供相应的 SQL 语句（不像 MySQL 可以直接使用 SQL 语句操作 XA 事务）。官方手册描述可以参考下面的链接，本节不再赘述。

[http://docs.oracle.com/cd/B28359\\_01/appdev.111/b28424/adfns\\_xa.htm#BGBBCJJI](http://docs.oracle.com/cd/B28359_01/appdev.111/b28424/adfns_xa.htm#BGBBCJJI)

需要注意，各个主流数据库虽然名义上都提供了对于 XA 的支持，但如 Oracle、PostgreSQL、MySQL 都是把其作为单机数据库支持分布式的一个补充<sup>①</sup>，而没有把 XA 上升到一个完整的分布式数据库的分布式事务处理的主要技术地位，所以本质上这些数据库都是单机数据库。只有在如 3.1.5 节所介绍的 Informix 的事务模型中，可以看到 Informix 把 XA 整合到了事务处理模块中，提升了 XA 机制在整个事务处理中的地位（不管是本地事务还是分布式事务处理都要经 XA 接口），但是 Informix 还不是一个真正地分布式数据库。

如下是 Oracle 对于 XA 的一个使用示例：

在 Client 端的应用中，可以使用库函数启动事务和结束事务：

```
tx_begin();           //开始一个事务
tpm_service("Service1"); //对Service 1进行操作
tpm_service("Service2"); //对Service 2进行操作
tx_commit();          //提交一个事务
```

而在数据库服务器 Server 1 和 Server2 中实现如下类似逻辑：

```
Service1() {
    // Get service specific data...
```

① 用法上，都是从应用角度出发，由应用主动控制事务；内核代码里，除 Informix 外，PostgreSQL、MySQL 只是把 XA 作为实现 2PC 的主要的部件，作为事务提交阶段的补充而实现的。所以说“没有把 XA 上升到一个完整的分布式数据库的分布式事务处理的主要技术地位”。

```
EXEC SQL UPDATE ....;
// Return service status back to the client...
}
Service2() {
// Get service specific data...
EXEC SQL UPDATE ....;
...
// Return service status back to client...
}
```

6.2 Oracle的封锁技术

Oracle 的并发控制技术，使用了 SS2PL 和 MVCC 技术。本节讨论封 Oracle 锁的基础知识，然后在 6.3 节讨论 MVCC 相关技术。

总的来看，使用了封锁技术的数据库系统，通常都会实现元数据锁和用户数据锁这两种类型的锁，如图 6-1 左部分内容所示，元数据锁支持 DDL 类型的并发操作，用户数据锁支持 DML、DQL 类型的并发操作。如 PostgreSQL、MySQL、Oracle 都是这样的。

如图 6-1 右部分内容所示，对应的是 Oracle 系统的元数据锁和用户数据锁的具体类型，如下细述这些类型的内容。

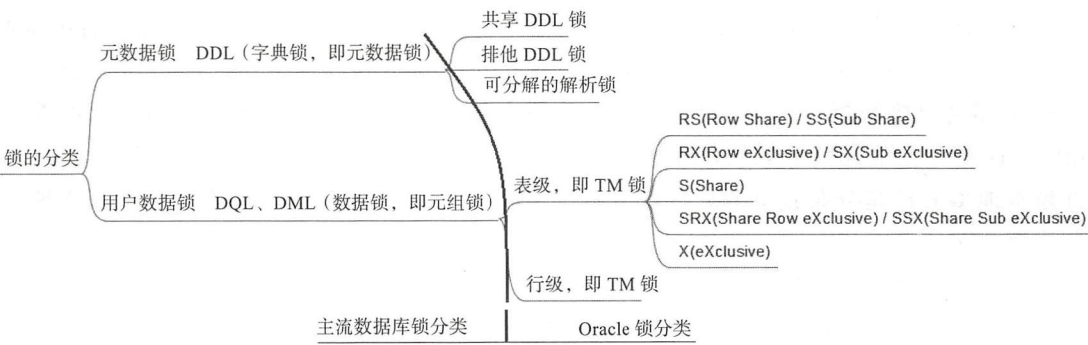


图 6-1 锁分类图

6.2.1 元数据锁的级别

元数据锁，是保护元数据如表对象不被并发操作破坏一致性，这对应的是 DDL 类型的操作。施加了元数据锁，能防止另一个用户 ALTER 或 DROP 被加锁的对象。

元数据锁只能自动施加，不能像“LOCK TABLE...”这样为对象显式施加锁。

Oracle 的元数据锁，分为三类，如表 6-1 所示。

表 6-1 Oracle 的 DDL 类型的锁

类型	加锁条件 / 原因	特征
Share DDL Locks	执 行 AUDIT、GRANT、REVOKE、COMMENT、CREATE PROCEDURE、CREATE FUNCTION、CREATE PACKAGE 等 DDL 语句	1) 不能防止类似的 DDL 语句或任何 DML 语句操作对象, 但能防止另一个用户 ALTER 或 DROP 被加锁的对象 2) 在 DDL 语句执行期间, 该 DDL 锁一直保持, 直到发生一个隐式的提交
Exclusive DDL Locks	执 行 ALTER TABLE、DROP TABLE、DELETE FROM TABLE、TRUNCATE TABLE 等 DDL 语句	完全互斥任何锁, 不管是 DDL 类还是 DML 类
Breakable parse lock	SQL 共享池里的对象, 处于 SQL 语句或 PL/SQL 的 Parse 阶段时, 被施加本锁	用来保护 SQL 共享池里的各个语句或 PL/SQL 对象

### 6.2.2 用户数据锁的级别

用户数据锁主要包括 TM 锁和 TX 锁, 其中 TM 锁称为表级锁, TX 锁称为行级锁。

#### 1. 表级锁

对于表级锁, Oracle 使用 TM 表示, 提供了五种粒度的锁, 分别是:

- ❑ Row Share (RS): 行共享表级锁, 又称为 subshare table lock (SS), 表示已经在表上锁定了表中的元组, 打算更新那些行, 但是否执行更新却不一定。
- ❑ Row Exclusive Table Lock (RX): 行排它表级锁, 又称为 subexclusive table lock (SX), 表示事务持有本锁的事务, 已经更新过元组, 或者是通过 “SELECT ... FOR UPDATE” 施加的 RX 锁。此时, 对于同一个表, INSERT、UPDATE、DELETE 或者 LOCK ROWS 等操作, 是不被互斥的。
- ❑ Share Table Lock (S): 共享表级锁, 持有此锁允许其他并发的事务在同一个表上执行查询操作 (但不可执行 SELECT ... FOR UPDATE)。
- ❑ Share Row Exclusive Table Lock (SRX): 共享行排它表级锁, 又称为 share-subexclusive table lock (SSX), 同一时间, 对于同一个表对象, 只能有一个事务可以获取此锁。但本锁不互斥对同一个表对象的查询操作 (但不可执行 SELECT ... FOR UPDATE), 互斥更新操作。
- ❑ Exclusive Table Lock (X): 排他表级锁, 互斥任何 DML 的操作。

这五种粒度的锁, 遵循表 6-2 所示的相容性矩阵 (一个单元格中的 Y 表示允许对另外并发的任务授予新申请的锁), 在并发的任务之间, 持锁者 (Granted Mode, 已经授予的锁) 允许或排斥后发出施加锁的请求 (Requested Mode, 正申请的锁)。

Oracle 可以用 LOCK TABLE 语句显式施加表级别的 TM 锁。

LOCK TABLE 语句的语法是:

```
LOCK TABLE [schema.]table_name IN LOCK_MODE MODE [NOWAIT];
```



LOCK\_MODE 表示锁的模式，取值是：SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE。对于常规的一些 SQL 操作，Oracle 施加的表级锁如表 6-3 所示。

表 6-2 Oracle 表级锁的相容性矩阵表（不同的事务间新锁的申请）

	Granted Mode, 已经施加的锁					
	N	SS	SX	S	SsX	X
Requested Mode 正申请的锁	SS	Y	Y	Y	Y	Y
	SX	Y	Y	Y		
	S	Y	Y		Y	
	SSX	Y	Y			
	X	Y				

表 6-3 Oracle 常见 SQL 操作表锁施加的表

SQL 语句	表锁模式
SELECT...FROM table ...	none
INSERT INTO table ...	RX
UPDATE table ...	RX
DELETE FROM table ...	RX
SELECT ... FROM table FOR UPDATE OF ...	RS
LOCK TABLE table IN ROW SHARE MODE	RS
LOCK TABLE table IN ROW EXCLUSIVE MODE	RX
LOCK TABLE table IN SHARE MODE	S
LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE	SRX
LOCK TABLE table IN EXCLUSIVE MODE	X
	RS: row share
	RX: row exclusive
	S: share
	SRX: share row exclusive
	X: exclusive

2. 行级锁

对于行级锁，Oracle 使用 TX 表示，锁定表中的一行数据。行级锁封锁遵循表 6-4 中的定义。

在执行 INSERT、UPDATE、DELETE、MERGE、SELECT...FOR UPDATE 语句时，Oracle 自动在被操作的表上加 TM 类型的锁，对于被操作的数据，会自动在被操作的行上加行级别的 TX 类型的锁。

对于 DQL 语句，Oracle 不施加任何锁（Oracle 的行级锁，只有 X 锁定模式，没有 S 锁定模式）。对于只读属性的事务，使用 MVCC 和快照与回滚段结合的技术，确保读操作不加锁时通过读取数据的前映像，就能读到一致的数据。

SELECT...FOR UPDATE 语句施加的锁与 UPDATE 语句施加的锁相同：一个是行级别的 EXCLUSIVE 锁、一个是表级别的 ROW EXCLUSIVE 锁。

Oracle 提供了一个特性，允许在 SELECT...FOR UPDATE 语句中提高并发度。即对于 SELECT...FOR UPDATE OF column\_list，有多个表被锁定时，可以指定要锁定的是哪几张表，当表中的列没有在 FOR UPDATE OF 后面出现时，这张表就不会被锁定，其他用户则可以对这些表的数据进行 UPDATE 操作的。

在 ORACLE 中，锁的数量不受限制且不会自动升级（有些数据库中，某个表上的行级锁达到一定数量后，这些行级锁就会被升级为该表上的表级锁，如 Informix），Oracle 认为锁的升级会为死锁检测带来困难，不提供锁升级的功能。

而 InnoDB 的元组级锁——记录锁，是加载在索引项的，记录锁被独立存放在内存锁表中，死锁检测的时候，会通过这样的内存锁表进行。但是，Oracle 和 PostgreSQL 的元组级锁，是施加在元组存储层的，Oracle 对于行锁施加在数据块上即页面上，PostgreSQL 对于行锁的施加在则是元组的元组头上（隐藏的系统字段）。

Oracle 提供了自动的死锁检测功能，发现死锁后，会选择一个事务进行回滚以消除死锁。

表 6-4 行级锁相容性矩阵表

	修改时申请 X 锁		读取时申请 S 锁		作用		
	操作结束 释放	事务结束 释放	操作结束 释放	事务结束 释放	避免 Lost Update	避免 Dirty Reads	避免 Nonrepeatable (fuzzy) reads
一级锁定协议 <sup>①</sup>		是			是		
二级锁定协议 <sup>②</sup>		是	是		是	是	
三级锁定协议 <sup>③</sup>		是		是	是	是	是

- ① 事务 T 在修改数据项 R 之前必须先对其加 X 锁，直到事务结束才释放。
- ② 事务 T 在修改数据项 R 之前必须先对其加 X 锁，直到事务结束才释放。事务 T 在读取数据项 R 之前必须先对其加 S 锁，读完后方可释放 S 锁。
- ③ 事务 T 在修改数据项 R 之前必须先对其加 X 锁，直到事务结束才释放。事务 T 在读取数据项 R 之前必须先对其加 S 锁，直到事务结束才释放。

6.3 MVCC技术

对于 MVCC 技术的理解，存在着较多的误区。而要想彻底理解 MVCC 技术，还需要依据以下各节的思路，深入探索。

6.3.1 MVCC 的历史

MVCC，多版本并发控制（Multiversion concurrency control）技术。

最早提出 MVCC 的，是 1978 年的《Naming and synchronization in a decentralized computer system》。之后在 1981 年，论文《Concurrency Control in Distributed Database



Systems》详细描述了 MVCC 技术，但是其描述的 MVCC 技术是基于时间戳的 MVCC，我们摘抄一段如下：

#### 4.3 Multiversion T/O

For rw synchronization the basic T/O scheduler can be improved using **multiversion data items** [REED78]. For each data item  $x$  there is a set of R-ts's and a set of (W-ts, value) pairs, called versions. The Rts's of  $x$  record the timestamps of all executed dm-read( $x$ ) operations, and the versions record the timestamps and values of all executed dm-write( $x$ ) operations. (In practice one cannot store R-ts's and versions forever; techniques for deleting old versions and timestamps are described in Sections 4.5 and 5.2.2.)

在这篇论文中，提出基于时间戳的 MVCC 技术，之后又有人提出基于封锁的 MVCC 技术，这些技术细节，请参阅第 2 章。

1995 年，《A Critique of ANSI SQL Isolation Levels》发表，批评了只以封锁技术为并发访问控制技术做基础、且只定义了三种读异常四种隔离级别的 SQL 标准，提出如图 6-2 所示的更多的隔离级别和更多的数据异常现象。

其中，“Snapshot isolation”，就是现在 PostgreSQL、InnoDB、Oracle 等所使用的 MVCC 技术的主干部分（实则基于多版本），但是此技术不能实现可串行化隔离级别。

到了 2008，《Serializable isolation for snapshot databases》发表，才使得 PostgreSQL V9.1 使用“Serializable Snapshot Isolation”技术实现了可串行化隔离级别。

但是，InnoDB 使用“Snapshot isolation”只实现了已提交读和可重复读两种隔离级别，而 Oracle 宣称自己使用的“Snapshot isolation”为“serializable”，但实际上这不是一个真正地可串行化技术，如 6.3.3 节所示，Oracle 只是部分地实现了“serializable”。但是，“Snapshot isolation”这个词也造就了“快照隔离级别”和“SQL 标准的四种隔离级别”的混淆，使得很多初学者分不清楚“快照隔离级别”和“SQL 标准的四种隔离级别”的区别。

Table 4. Isolation Types Characterized by Possible Anomalies Allowed.								
Isolation level	P0 Dirty Write	P1 Dirty Read	P4C Cursor Lost Update	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	A5A Read Skew	A5B Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

图 6-2 多种隔离级别和数据异常现象之间的关系





所以,“Snapshot isolation”是宏观的 MVCC 技术的一个技术变种,“Snapshot”(快照)技术和多版本技术结合的狭义的 MVCC,才是 PostgreSQL、InnoDB、Oracle、SQL Server 等这些数据库所谓的 MVCC。

## 6.3.2 深入理解 MVCC

### 1. MVCC 解决了什么样的问题

MVCC 技术,是并发访问控制的核心技术之一,在数据库中,用于防止用户表数据被并发的事务访问出现数据不一致的问题。这里需要区分的是用户表的元数据和表数据。

对于数据库系统而言,元数据是使用 DDL 语句操作的,通常使用封锁并发访问控制技术来互斥并发、确保元数据一致,而加锁的粒度通常很粗,多是排它锁、完全抑制并发。但如上节所述,Oracle 却把 DDL 加锁的粒度分为三种,增加了并发度。

而用户数据,固然可以使用封锁并发访问控制技术来互斥并发、确保元数据一致,如 Informix 就是完全依赖封锁(读操作加锁)实现了可串行化隔离级别。但是,这样的并发度太低(参见 2.2.1 节“5. 锁的并发度的问题”),数据库的性能会差,因为读操作会阻塞写操作,写操作会阻塞读操作,写操作也会阻塞写操作。MVCC 技术,通过对用户的表数据,即元组进行“分身”处理,把一个元组按照其生存状态(出生中<sup>⊖</sup>、活着<sup>⊖</sup>、死去<sup>⊖</sup>、正在经历凤凰涅槃般的生死历程<sup>⊗</sup>等)和阶段生命期(某个版本的诞生和死亡时间段)进行区分。就这样对于一个数据项,就会在时间的长河中有多个阶段,每个阶段对应一个版本,每个版本供同样处于时间长河的生命周期不同的事务去读或写,而此刻,多个版本就是一个静态的存在。

静态存在的版本,其状态和生命周期会被动态的事务所修改,并发的存在,使得有多个动态的事务可能同时读写同一个数据项的不同版本、甚至是同一个版本。这时,就需要 MVCC 技术识别某个版本对于某个事务是否是“可见”的,可见的含义有两个:

- 一是本事务创建、修改过的版本,一定对于本事务可见。
- 二是非本事务操作的版本,只有这个版本的状态是已经提交的状态,才对其他事务可见。这句话首先暗含了“已提交读”的含义,这样就会避免脏读异常现象。其次,判断“版本的状态是已经提交的状态”却是一个复杂的过程,不同数据库有着不同的技术实现,如第 9 章详细讲述了 PostgreSQL 的可见性判断的方法、第 12 章详细讲述了 InnoDB 的可见性判断的方法等。

静态的版本和动态的并发事务结合,通过抑制并发操作,使并发的的事务能被确保符合可串行化调度,就是 MVCC 技术的核心所在。

⊖ INSERT或UPDATE操作造成的一个过程状态。

⊖ INSERT或UPDATE操作造成的一个结果状态。

⊖ UPDATE或DELETE操作造成的一个结果状态。

⊗ UPDATE操作造成的一个过程状态,使得旧版本历经死去、新版本历经诞生。



因为多版本的存在,不同的并发事务,总是在操作不同的版本,所以读-写互不阻塞,写-读也互不阻塞。只有写-写操作并发的时候,存在冲突,解决的方式有两种,要么是先更新者优先(后更新的被回滚),要么是先提交者优先(后提交的被回滚,PostgreSQL即如此。其他主流的数据库中,尚没有基于快照和多版本实现的MVCC实例实现了可串行化隔离级别)。

到此时此刻,需要注意的是,我们讨论的前提,都是在保证一致性的前提下进行的,即形式上是为了实现可串行化隔离级别的。至于其他的隔离级别,牺牲了数据的一致性,提高了数据库的性能,把避免出现数据不一致的问题抛给了基于数据库做应用开发的程序员,实属无奈或无责?

## 2. MVCC 与锁的关系

既然MVCC是解决元组一级数据一致性和并发操作的技术,那为什么还要使用行锁? PostgreSQL、InnoDB、Oracle都在元组一级使用了行级锁,这究竟为什么呢?

对这个问题的思考,可以帮助我们更深入地理解MVCC技术和封锁技术的不同。如前所述,MVCC解放了读-写、写-读的并发冲突,路是从低并发度向高并发度方向走去的;而锁的施加,抑制了并发操作,路是从高并发度向低并发度行进的。两者从两个点出发,相向而行,必然会遇到一起、携手共勉的。这固然是个比喻,读者大可一笑了之,但是,其中的含义,我们还是要明言的:

□一是MVCC通过多版本,解决了并发操作对“同一个对象”(同一个数据项的不同版本)的“读-写、写-读”竞争问题,但是没有解决“写-写”竞争。

□二是封锁技术,不仅抑制了“读-写、写-读”并发,更是抑制了“写-写”并发。

所以,通过对元组的并发访问控制(或者说对于DML类操作的并发访问控制),这两种技术就有了结合点,各用其长,解决“写-写”并发造成的竞争问题,这就是PostgreSQL、InnoDB、Oracle等主流数据库都采用封锁技术和MVCC相结合的原因,用锁来解决元组级的“写-写”并发,这样的策略,可以称之为“先更新者获胜”。

另外,InnoDB和PostgreSQL、Oracle等还有不同之处,就是InnoDB的记录锁不是真正的元组上的锁,而是施加在索引的记录上的索引记录锁,所以InnoDB的多版本,实则是索引记录项存在多个版本在回滚段中。

## 3. 读一致性

Oracle中使用MVCC实现多版本读一致性(“Multiversion Read Consistency”)的技术,一致为Oracle迷们所津津乐道。这样的技术,却是大大提高了并发度,使得并发操作对“同一个对象”(同一个数据项的不同版本)的“读-写、写-读”竞争问题不复存在。

但是,这个技术,不是Oracle的独家秘籍,实现了MVCC技术的诸如PostgreSQL、InnoDB都存在一样的效果。换句话说,使用了快照和多版本的MVCC技术,都会拥有同样的能力。

在PostgreSQL中,没有一个特定的名词与“Multiversion Read Consistency”对应,但



是多种隔离级别下的快照配合数据页元组级的多版本均存在这样的能力。而 InnoDB 对应有一个名称，叫做“Read View”，异曲同工。

所以，Oracle 提供的“Transaction-Level Read Consistency”“Statement-Level Read Consistency”也不是什么新鲜东东，其实就是可串行化隔离级别、已提交读隔离级别下 MVCC 技术的应用。只是需要注意的是，不同数据库的“Read Consistency”实现方式不完全相同，Oracle 使用了回滚段技术来实现旧版本的一致性读取，这点类似 InnoDB。而 PostgreSQL 纯粹是依赖多版本和快照实现了一致性读取。

### 6.3.3 Oracle 的 MVCC

Oracle 的 MVCC 技术和存储层的数据结构密切相关，因此我们先从存储结构入手来了解 Oracle 的 MVCC 技术，但是本节不会详细深入到具体的细节中（更多详情请参考 Oracle 的《DSI402 Space and Transaction Management》等相关资料），然后讨论 MVCC 技术的实现方式。

特别声明，如下的一些图片并非原创，分别源自：

- 《DSI402 Space and Transaction Management》：图 6-3、图 6-4、图 6-6、图 6-7、图 6-11。
- 《Oracle Core Essential Internals for DBAs and Developers》：图 6-5、图 6-9、图 6-10、图 6-12。
- Oracle 官方手册：图 6-8。

#### 1. 元组结构

Oracle 的一条元组，结构如图 6-3 所示。其中，  
从左至右：

Row flag (1)	Lock byte (1)	# of cols (1)	Row SCN (6)	Column length (1 or 3)	Column data
--------------------	---------------------	---------------------	----------------	------------------------------	----------------

图 6-3 Oracle 元组结构体图

- 第一个字节，Row flag 是一个元组状态标志位，表示元组是否被删除。
- 第二个字节，Lock byte 是一个锁标志位，0 表示没有施加锁，1 表示有事务施加了锁。
- 第三个字节，列的个数。
- Row SCN (system commit number)，格式为“0xFFFF.FFFFFFFF”，共六个字节，是一个元组的 SCN 号。这样的 SCN 号可以和时间值互换<sup>①</sup>表示，表明其本质上就是一个时间标识，用以表明元组被创建或修改的时间点。
- 一条元组存放到一个页面的“Data Layer”中。

#### 2. 页面结构

Oracle 的一个物理块，结构如图 6-4 所示，主要包括：

- Cache Layer：主要存放三部分数据：
- 旧的 SCN 相关内容（前象，用于恢复使用），包括：data block、undo block、RBU

① 互换的方法为：SELECT DBMS\_FLASHBACK.GET\_SYSTEM\_CHANGE\_NUMBER scn1, TO\_CHAR(SCN\_TO\_TIMESTAMP(28607851), 'yyyy-mm-dd hh24:mi:ss') chr, TIMESTAMP\_TO\_SCN(SCN\_TO\_TIMESTAMP(28607851)) dt FROM DUAL;





hdr block。

❑ 新的 SCN 相关内容 (后象, 被修改后的内容), 也包括新的: data block、undo block、RBU hdr block。

❑ Change vectors: REDO 日志中记录的顺序的数据变更信息。

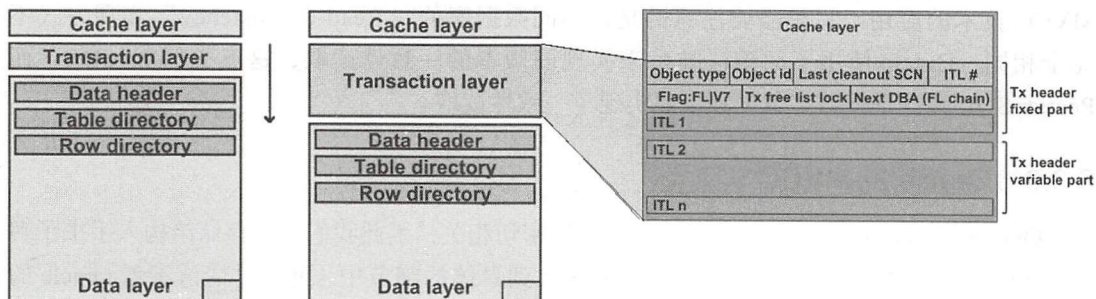


图 6-4 Oracle 数据页面结构

❑ Transaction Layer: 事务操作的相关信息。又分为两部分:

❑ 第一部分大小固定, 包括事务头信息和一个 ITL (interested transaction list)。

❑ 第二部分是变长的, 包括零到多个 ITL, 如图 6-4 左图可变迁为中图样式, 即变迁为多个 ITL。

❑ 每个事务执行 DML 操作 (INSERT、DELETE、UPDATE), 就分配一个 ITL<sup>①</sup>。

❑ 每个 ITL 有 24 个字节, 如图 6-5, 包括有:

❑ 事务拥有 ITL 的标识符, 表示为: usn#.slot#.wrap#, 即事务 ID, Xid, xid = Undo Segment Number + Transaction Table Slot Number + Wrap 也即: 回滚段号+事务槽编号+序号 (同一个事务可能具有多个 SCN, 实际上每一个 DML 操作都有一个 SCN)。如 “0x0008.015.00000006” 是一个事务 ID, 其中 Undo Segment 是 8 (0x0008)。而十进制的 21 对应的 0x015 表示来自 “the Transaction tTable” 的槽为 21 “undo block”。“0x00000006” 则是第 6 次被复用。

❑ 记录在回滚段中的地址 (Undo block address, UBA), 表示为: DBA.seq#.rec#。其中, DBA 的含义是 Data Block Addresses, 被修改的数据块位置。

❑ 事务在本页面上持有的锁的个数。

❑ 如果事务已经提交, 则记录事务提交的 SCN 值 (事务提交的标志)。如果事务还没有提交, 记录本事务拥有的空闲空间信息。

❑ Data Layer: 存放数据的区域。可以存放多个如前所述的元组。

① The database uses the ITL to determine whether a transaction was uncommitted when the database began modifying the block. Entries in the ITL describe which transactions have rows locked and which rows in the block contain committed and uncommitted changes. The ITL points to the transaction table in the undo segment, which provides information about the timing of changes made to the database



Column	Description
Itl	The array index for the list. The number isn't physically stored in the block; it's generated by the code that does the dump. This value is used in the <i>lock byte</i> (lb:) for a row to show which transaction has locked the row.
Xid	The transaction id of a recent transaction that has modified this block. The format is undo segment . undo slot . undo sequence number.
Uba	The undo record address—including the sequence (or incarnation) number—of the block of the most recent undo record generated by this transaction for this block. The format is Absolute block address . block sequence number . record within block. (The “b” in the label suggests byte or block, but neither of those interpretations is quite accurate.)
Flag	Bit flag identifying the apparent state of this transaction: ----: active (or “never existed” if every field in the Xid is zero). --U-: Upper bound commit (also set during “fast commit”). C---: Committed and cleaned out (all associated lock bytes have been reset to zero). -B--: May be relevant to the recursive transactions for index block splits. I have seen comments that this flag means the UBA will point to a record holding the previous content of the ITL entry, but I have not managed to confirm this. ---T: I have seen comments that this means the transaction was active during block cleanout, but I have not managed to confirm this.
Lck	Number of rows locked by this transaction in this block.
Scn/Fsc	Depending on the Flag, the commit SCN or the number of bytes of free space that would become available if this transaction committed (Free Space Credit).

图 6-5 Oracle 的 ITL 含义

### 3. 事务和锁

Oracle 的物理页面上有 “Transaction Layer”，所以事务操作的单位是页面而不是元组。元组上有 “Lock byte”，所以锁信息记载的载体是元组而不是数据页面。

如图 6-6 所示，两个并发事务 Tx1 和 Tx2，事务 Tx1 操作 row2 和 row3 元组，事务 Tx2 操作 row1 元组，各自加锁成功，锁施加在元组头的 “Lock byte”，旧值 0 被置为 1，表示有锁存在。

如图 6-7 所示，事务 Tx2 紧接着想操作 row2 元组，但是 Oracle 元组级锁提供的只有排它锁模式，事务 Tx2 启动一个称为 “block cleanout” 的操作（本质上是确认事务 Tx1 是否已经提交<sup>①</sup>），如果 “block cleanout” 操作执行后事务 Tx1 还是处于 active 状态，则判定存在冲突。

之后事务 Tx2 只能等待（TX enqueue）。而事务 Tx3 启动后，打算操作 row1 元组，发现 row1 元组上也存在锁，同样需要进行 “block cleanout” 的操作，因冲突存在只能等待。

结合元组结构和 ITL，我们可以知道：Oracle 通过 “Transaction Layer + ITL + lb(row header)” 实现了行级锁。一个事务拥有一个 TM lock（表锁）和若干 TX lock（行级锁），元组所在的每个数据块只需要 1 个 ITL。

① 确认过程是通过构造 “block list state object” 进行。



Row Locking: Success

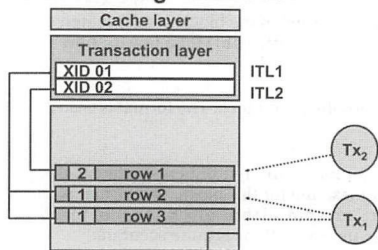


图 6-6 Oracle 锁与锁冲突 (一)

Row Locking: Conflict

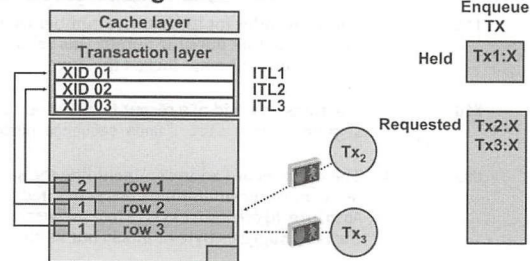


图 6-7 Oracle 锁与锁冲突 (二)

#### 4. 数据页的事务处理逻辑

有了上述的物理结构基础，我们可以探索一下事务在数据层操作的逻辑过程。一个事务执行的时候，数据页上的各个层次之间的关系如图 6-8 所示。

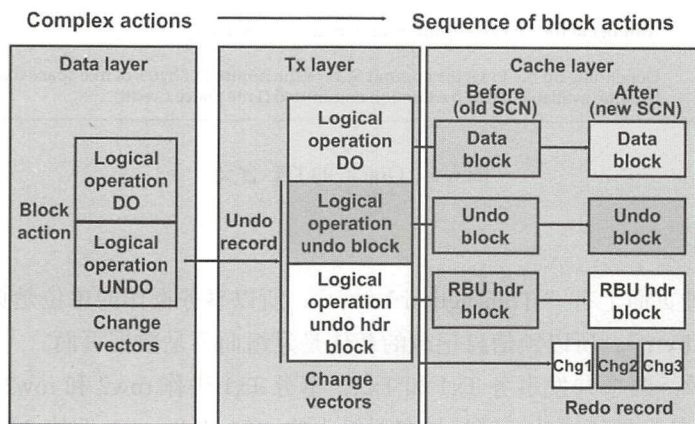


图 6-8 Oracle 存储结构与事务的关系图

对于数据层 Data Layer 的修改，相关信息记载在事务层 Tx Layer 和缓冲层 Cache Layer 中。事务层 Tx Layer 存放事务相关的一些信息，事务层即前述的 ITL。

对于通常的 SQL 操作，涉及修改数据页的（DDL、DML 都会修改数据页），会生成操作序列，把过程中的引发数据页变化的内容的一个序列，先后写到 REDO 日志，以备故障发生时再次执行；而数据的前象部分，会被以页面为单位，存储在回滚段中；而写回滚段的操作，修改了回滚段这个对象，此修改动作也需要在故障恢复过程中被重新执行，所以对于回滚段的修改的可串行操作，也要记录到 REDO 日志，这就涉及回滚段的某些数据页的页头、修改前的回滚段某些数据页所在的页块信息、逻辑操作的信息（对应到图 6-8 中顺序为：Chg1、Chg2、Chg3）。





## 5. 多版本与回滚段

Oracle 的回滚段，有四个作用：

- ❑ 回滚一个活动的事务。
- ❑ 恢复一个被中止的事务。
- ❑ 提供读一致性。
- ❑ 提供逻辑闪回功能。

为了实现上述四个功能，Oracle 提供了回滚段的功能。回滚段是一个被反复复用的一个数据结构（因复用冲掉了旧的信息所以导致“snapshot too old”，是一个宝贵的数量有限的资源，包括多个 Undo Segment（图 6-10）。每一个 Undo Segment 又包括多个 Undo block（图 6-11）。图 6-8 显示了利用回滚段读取数据时，如果发现需要读取历史版本的数据，就从回滚段中获取旧数据的“一致性读”的基本过程。读数据的时候，读操作对应的 SCN 值是 10023，所以只能读到小于此值的数据，当读到 SCN 值为 10024 的数据时，因 10024 大于 10023，则去回滚段中寻找旧数据读取，这就是读一致性的基本含义与实现方式。

对于每一个具体的版本，同其他实现了 MVCC 技术的数据库一样，也需要进行元组的可进性判断，基本原则是：

- ❑ 本事务自己产生的版本，本事务可见。
- ❑ 读非本事务产生的版本，必须是已经提交的版本。
- ❑ 读非本事务产生的版本，必须是已经提交的版本中最新的。

Undo Segment 的结构如图 6-9 所示。

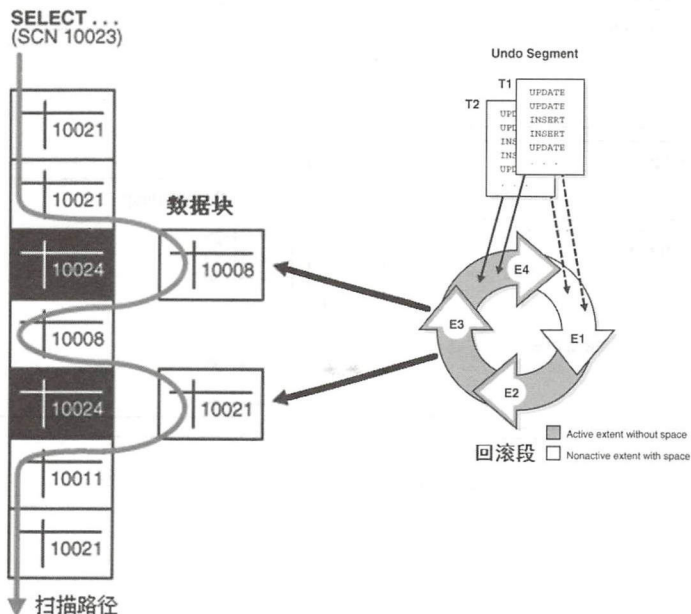


图 6-9 读一致过程图

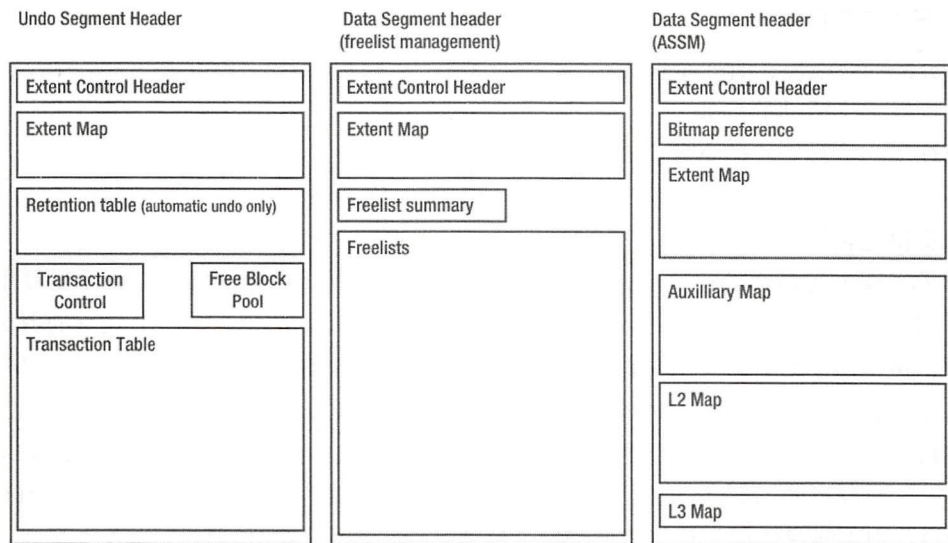


图 6-10 Oracle 的 UNDO Segment 结构图

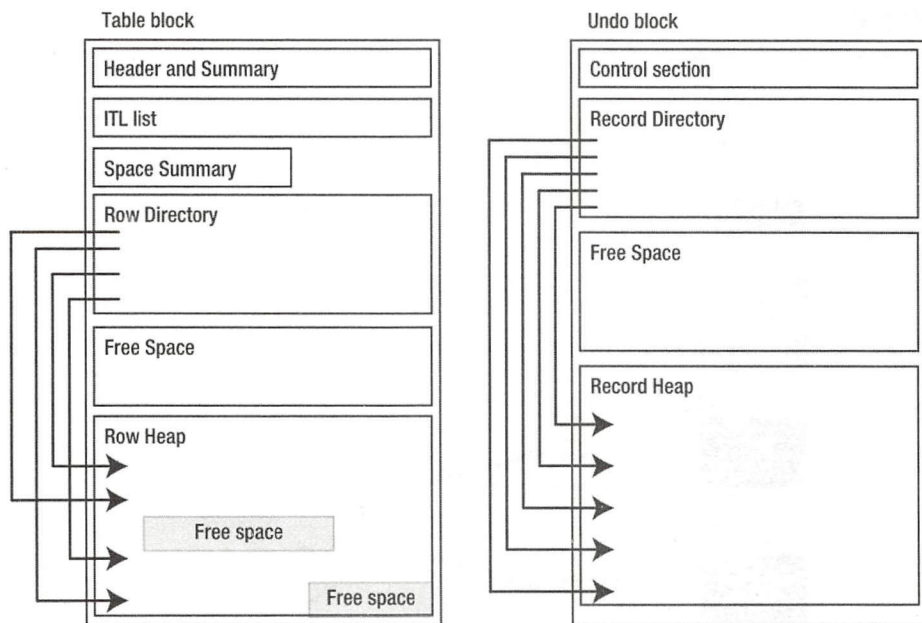


图 6-11 Table Block 和 Undo Block 结构示意图

## 6. 小结

引用《Oracle Core Essential Internals for DBAs and Developers》的一段话，很精彩，

概括了 Oracle 的存储层物理结构和事务的关系，本书不再赘述，请静静地多读几遍吧：

There are two key structures in the database that make it possible for Oracle to handle transaction processing and read consistency efficiently: the interested transaction list (ITL) that appears in each data block and lists recent transactions that affected that block, and the transaction table that appears in the segment header block of each undo segment and lists recent transactions that affected the database.

An ITL entry records a transaction ID (xid:), an undo record address (uba:), and a commit SCN. The commit SCN tells Oracle if (and when) the transaction committed. If the commit SCN is not available, then the transaction ID identifies a transaction table slot with sequence number, and this information allows Oracle to check the state of the transaction and when (if) it committed. If your session needs to hide the changes made by that transaction, it can use the undo record address to find the start of a chain of undo records that tells it how to reverse the changes made by that transaction to that data block.

A transaction table slot records the state of a transaction, the address of the last undo block that it wrote to, and (when committed) the commit SCN. Because there aren't many transaction table slots in a database, they are continuously reused, so the slot has a wrap# number counting the number of times it has been used, and this also becomes part of the transaction ID. If a transaction has to be rolled back, the pointer to the last undo block allows Oracle to find the last undo record created by the transaction, and each undo record points to the previous undo record for its transaction, so the undo records can be visited and applied in the opposite order to the order in which they were created.

Transaction table slots can be overwritten fairly quickly, which means the commit SCN for a transaction would be lost if Oracle didn't have a mechanism to preserve information from the transaction table slot before reusing it. Each undo segment header has a transaction control section summarizing the use of the transaction table. As a transaction table slot is overwritten, its commit SCN is written into the transaction control along with the address of the first undo record of the transaction that has just acquired the slot. The previous information in the transaction control is then written into the first undo record of the new transaction, and this effectively builds a linked list of undo records (the “first record” of each transaction) that can be used to roll the transaction table back to an earlier point in time.

Any time that Oracle is following a list of pointers through the undo segment and arrives at an undo block with the wrong seq: (i.e., incarnation) number, you will get an Oracle error ORA-01555, “snapshot too old,” because the block you wanted to see has been reused.



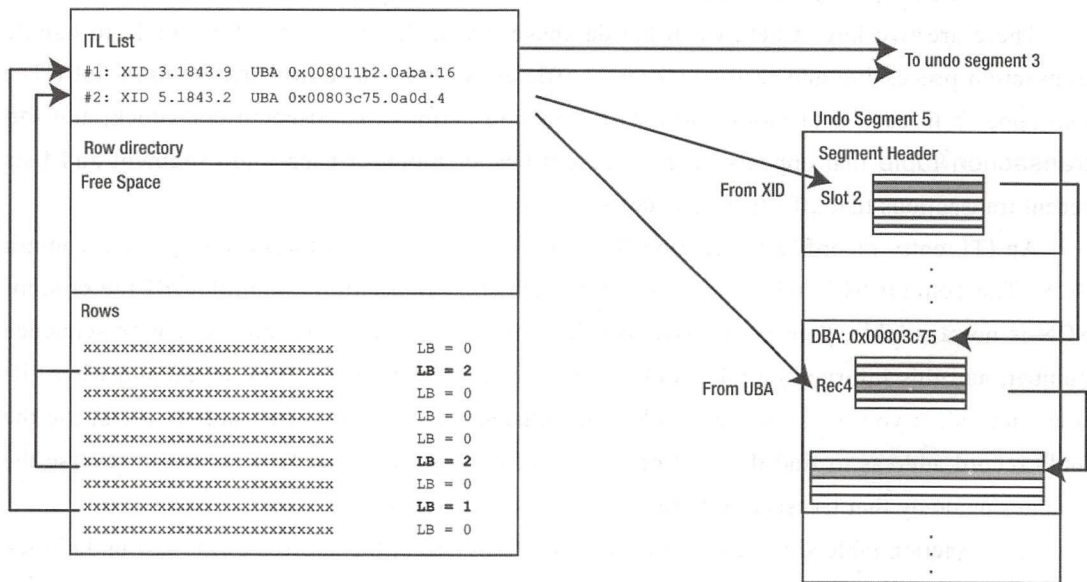


图 6-12 数据页与回滚段的关系图

## 7. 设计思想

Oracle 利用 MVCC 实现了“读-写”、“写-读”两种情况的互不阻塞，提高了并发度；利用 MVCC 技术消除了幻象数据异常（这点同 PostgreSQL）。但是，Oracle 并没有实现真正的可串行化调度，尽管设置隔离级别时可以指定可串行化隔离级别，这是因为 Oracle 存在写偏序异常，这点可以参见 6.4.2 节。所以，Oracle 的隔离性的实现，只是一个“快照隔离”的实现，没有实现数据的“一致性”。

Oracle 事务处理的一大特色，就是事务处理和存储紧密关联，而其他数据库诸如 PostgreSQL 的事务虽然和元组的结构有关联，但仅仅限于在元组头上记录事务号这一点儿的信息，没有类似 Oracle 的 ITL 和事务表等这样的信息，所以事务和存储的关系处于松耦合状态。

Oracle 的 RAC 充分利用了事务和存储紧耦合的关系，为了保证读数据的一致性，直接在物理节点间传输数据块，不仅使得数据得到传播而且附着的事务信息也能被其他的物理节点获取，这样就便于实现分布式的事务处理。从这一点看，Oracle 的事务和存储紧密结合的设计，是有优势的。

但是，分布式数据库的设计，已经呈现多种模块分离的趋势，如 SQL 解析、SQL 优化、SQL 执行、事务管理、数据存储等趋向于松耦合，所以类似 Oracle 的事务和存储紧密结合的设计，未必是一个可借鉴的模式。

Oracle 对于快照和多版本的实现，没有更多资料能够帮助我们深入细节，但是类似地

我们可以阅读第7章到第9章 PostgreSQL 的并发控制技术的实现来对照、思考 Oracle 的实现。对于利用回滚段实现多版本和事务的回滚操作，则可以阅读第10章到第12章 InnoDB 的并发控制技术的实现来对照、思考 Oracle 的实现方式。这样我们就能比较完整地刻画出 Oracle 的并发控制技术的实现原理，并猜测其实现方式。

Oracle 的事务处理还有很多细节，读者可以参考相关资料，或者阅读 6.3.3 节初始推荐的资料，进一步深入研究，本章将不再做更进一步深入探讨。

## 6.4 隔离级别与数据异常

Oracle 支持三种隔离级别，但是这三种隔离级别却不完全等同于 SQL 标准定义的隔离级别。

### 6.4.1 Oracle 支持的隔离级别

Oracle 支持三种隔离级别，但是这三种隔离级别和 SQL 标准定义的四种隔离级别不尽相同，如表 6-5 所示。

Oracle 缺省的设置是 Read Committed 隔离级别，在这种隔离级别下，如果一个事务正在对某个表进行 DML 操作，同时另外一个并发事务对这个表的记录进行读操作，则 Oracle 会去读取回滚段中存放的老版本的记录，这样的方式被称为一致性读（read consistency，InnoDB 中也存在类似的概念。其实，基于快照的 MVCC 技术对元组进行并发访问控制的技术都可以实现类似的功能，即读和写互不阻塞，再如 PostgreSQL 亦如此）。

Oracle 的可串行化隔离级别，不是一个完全地可串行化实现。这是因为 Oracle 在可串行化隔离级别时，可能报告“snapshot too old”<sup>①</sup>这样的“因资源不足”而引发的错误。完整的可串行化实现，逻辑上不应受物理资源的限制，都应该实现可串行化调度而不是报告因资源不足引发的错误。比如，PostgreSQL 使用 SSI 技术实现了并发事务的可串行化调度，PostgreSQL 存放“谓词锁”的锁表位于共享内存中，其大小也是固定的，而并发事务的数量是无限的，但 PostgreSQL 不会因为用于判断是否符合可串行化调度的“谓词锁表”容量有限而报告资源不足这类的错误（当然，Oracle 通过报告错误迫使用户事务回滚这种设计也是一种折中方案），而是采用合适的技术完美地解决问题为“更好”（PostgreSQL 事务汇总技术的实现细节，参见 9.4 节）。

另外，Oracle 支持了“Read-Only”“隔离级别”，与其说其是一种隔离级别，倒不如说是“事务的属性”为好。基于快照技术的 MVCC，都容易实现一个只读的事务，这是因为一个事务只使用同一个快照即可做到只读事务的所有读操作都是读一致的，如 PostgreSQL

① 不管是哪种隔离级别，Oracle 需要获取数据的时候，可能需要从回滚段查找旧版本数据，但是回滚段空间有限，不可能存储全部旧版本数据，一些旧版本数据随着回滚段被重用而被丢弃，这样对于需要那些被丢弃的旧版本数据的事务而言，Oracle 则会报告“snapshot too old”。

有只读事务、InnoDB 有一致性读等相似概念。

Oracle 通过 MVCC 技术，解决了 SQL 标准规定的三种读数据异常现象，但是其可串行化隔离级别不是一个完整正确的可串行化实现，因为 Oracle 不能完全避免数据异常现象，比如 6.4.2 节在 Oracle 11g 版本测试验证，写偏序异常存在。

表 6-5 Oracle 隔离级别与 SQL 标准定义的数据异常表

SQL 标准名称	Oracle 定义	脏读	不可重复读	幻象	写偏序
READ UNCOMMITTED 未提交读	—	—	—	—	—
READ COMMITTED 已提交读	Comitted Read	避免	发生	发生	发生
	Read-Only	避免	避免	避免	—
REPEATABLE READ 重复读	—	—	—	—	—
SERIALIZABLE 可串行化	SERIALIZABLE	避免	避免	避免	发生

6.4.2 写偏序异常

Oracle 的可串行化隔离级别可以避免任何数据异常，这是很多人的认识，其实不然。如下使用 Oracle 11g 测试，可以看到，Oracle 存在写偏序异常。

1. 数据准备

```
CREATE TABLE dots (  
    id INT NOT NULL PRIMARY KEY,  
    color VARCHAR(20) NOT NULL  
);  
  
INSERT INTO dots VALUES (1, 'black');  
INSERT INTO dots VALUES (2, 'white');  
INSERT INTO dots VALUES (3, 'black');  
INSERT INTO dots VALUES (4, 'white');
```

2. 可串行化隔离级别

对于可串行化隔离级别，并发执行如表 6-6 中的命令。

表 6-6 隔离级别为“SERIALIZABLE”的并发表

Client1	Client2
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	



(续)

Client1	Client2
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新没有被阻塞。因数据量少，数据有很大的可能位于同一个页面，而并发操作依旧可以执行，说明并发应该是行级并发，不是页面级的
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
COMMIT; 成功	
	COMMIT; 成功
	SQL> SELECT * FROM dots; ID COLOR ----- 1 white 2 black 3 white 4 black 这说明，可串行化隔离级别下，写偏序异常存在

3. 已提交读隔离级别

对于已提交读隔离级别，并发执行如下命令，如表 6-7 所示（数据保持初始状态，即删除数据后，重新插入 4 条数据）。

表 6-7 隔离级别为“REPEATABLE READ”的并发表

Client1	Client2
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	
	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE dots SET color = 'black' WHERE color = 'white'; 更新成功	
	UPDATE dots SET color = 'white' WHERE color = 'black'; 更新没有被阻塞。

(续)

Client1	Client2
SELECT COUNT(*) FROM dots WHERE color = 'white'; count ----- 0 (1 row)	
COMMIT;	
	COMMIT; 注意: 此示例此时, 没有检测到写偏序异常, 提交成功
	ID COLOR ----- 1 white 2 black 3 white 4 black 这说明, 已提交读隔离级别下, 写偏序异常存在

6.5 本章小结

本章同第 3、4、5 章相似, 首先探讨了事务模型, 其次重点探讨了 Oracle 的并发访问控制技术, 包括封锁技术和 MVCC 技术, 并探讨了数据异常和隔离级别之间的关系, 这些讨论没有限于 Oracle 的范围, 而是涉及了 Informix、PostgreSQL 和 InnoDB, 用意在于形成一个对比, 以帮助读者更好地理解数据库的并发控制技术的原理, 感悟相同的原理下不同的实现差异。

## 第三篇

# PostgreSQL事务管理与并发控制源码分析

本篇讲述 PostgreSQL 事务管理和并发控制技术的实现技术，主要是从源码的角度，结合本书第一篇的理论基础部分，详细地进行关键代码的分析。

建议读者在阅读理解本篇的时候，既要结合第一篇的理论，又要对照第四篇的 InnoDB 的事务管理和并发控制实现技术的分析，进行对比掌握。



## 第 7 章

# PostgreSQL 事务系统的实现

本章讲述 PostgreSQL 的事务系统的实现。包括 PostgreSQL 事务管理的基本思想和实现方式，在实现方式中，重点讲述事务实现的数据结构和实现过程。

## 7.1 架构概述

PostgreSQL 的事务处理模块，是整个数据库的基石。其中，重要的组件，包括事务管理模块和并发访问控制模块。并发访问控制模块在第 8 章和第 9 章详细进行讨论。从本节开始，讨论事务的管理模块，这个模块实现了事务的整体模型，包括事务的提交和回滚以及子事务等。

本节先从整体上讨论 PostgreSQL 对事务管理的实现基础，包括事务管理涉及的文件及主要功能、事务管理的整体架构等。

### 7.1.1 事务和并发控制相关的文件

PostgreSQL 在“src/backend/access/transam”目录下存放了与事务和日志管理相关的一些文件，如表 7-1 所示。在“src/backend/storage/lmgr”目录下存放了与锁管理相关的一些文件，如表 7-2 所示。另外，还有一些分布在不同的目录中重要的文件，这些文件，参见表 7-3。

表 7-1 PostgreSQL 事务和日志管理相关文件表

文件名	功能
clog.c	PostgreSQL transaction-commit-log manager 记录事务提交或回滚的状态信息，并提供相应的管理功能
commit_ts.c	PostgreSQL commit timestamp manager

(续)

文件名	功能
commit_ts.c	存储每个事务的提交时间戳（与 clog 配套使用）
generic_xlog.c	Implementation of generic xlog records. Xlog 的辅助操作函数
multixact.c	PostgreSQL multi-transaction-log manager It is a fundamental part of the shared-row-lock implementation. 并发事务的管理
parallel.c	Infrastructure for launching parallel workers. PostgreSQL 9.6 新增并行计算功能，这个文件中的函数支持了并行计算
rmgr.c	Resource managers definition.
slru.c	Simple LRU buffering for transaction status logfiles. PostgreSQL 使用最近最少使用算法（Least Recently Used）管理事务状态缓存池
subtrans.c	PostgreSQL subtransaction-log manager. It is a fundamental part of the nested transactions implementation. 子事务的管理
timeline.c	Functions for reading and writing timeline history files. 用于对恢复过程的多个基于不同的恢复点进行恢复操作的事件进行管理
transam.c	postgres transaction log interface routines.
twophase.c	Two-phase commit support functions. PostgreSQL 支持 XA，即分布式事务的管理
twophase_rmgr.c	Two-phase-commit resource managers tables. 完成 2 阶段提交的辅助功能，如支持提交、回滚、故障之后进行恢复
varsup.c	postgres OID & XID variables support routines. 对事务号进行管理，如 GetNewTransactionId() 可以获得新的事务 ID
xact.c	top level transaction system support routines. 事务管理的主要操作
xlog.c	PostgreSQL transaction log manager. Redo-log, PostgreSQL 的重做日志；重做日志主要操作位于此文件，如刷除重做日志 XLogWrite()
xlogarchive.c	Functions for archiving WAL files and restoring from the archive. 重做日志的归档操作
xlogfuncs.c	PostgreSQL transaction log manager user interface functions. 重做日志的一些接口，为用户管理重做日志提供便利
xloginsert.c	Functions for constructing WAL records. 构造出重做日志，插入重做日志缓存区
xlogreader.c	Generic XLog reading facility. 从 Redo-log 中读出日志记录，解析格式。如 XLogReadRecord()
xlogutils.c	PostgreSQL transaction log manager utility routines. Redo-log 的一些辅助操作

表 7-2 PostgreSQL 锁管理相关文件表

文件名	功能
deadlock.c	死锁检测
lmgr.c	锁管理器
lock.c	事务锁相关的操作
lwlock.c	LWLock，轻量锁的相关操作
predicate.c	基于 MVCC 技术的 SSI 技术实现序列化隔离级别
proc.c	标记安全功能
spin.c、s_lock.c	spin 锁的实现

表 7-3 PostgreSQL 事务和并发控制相关的其他文件表

文件名	功能
snapmgr.c	快照相关管理的文件
tqual.c	元组可见性、可更新性的相关操作

7.1.2 事务相关的整体架构

PostgreSQL 的事务系统，贯穿在整个数据库系统中，不仅包括 SQL 语句在内存执行的阶段，而且包括在存储层如扩展文件块等处，如图 7-1 所示。

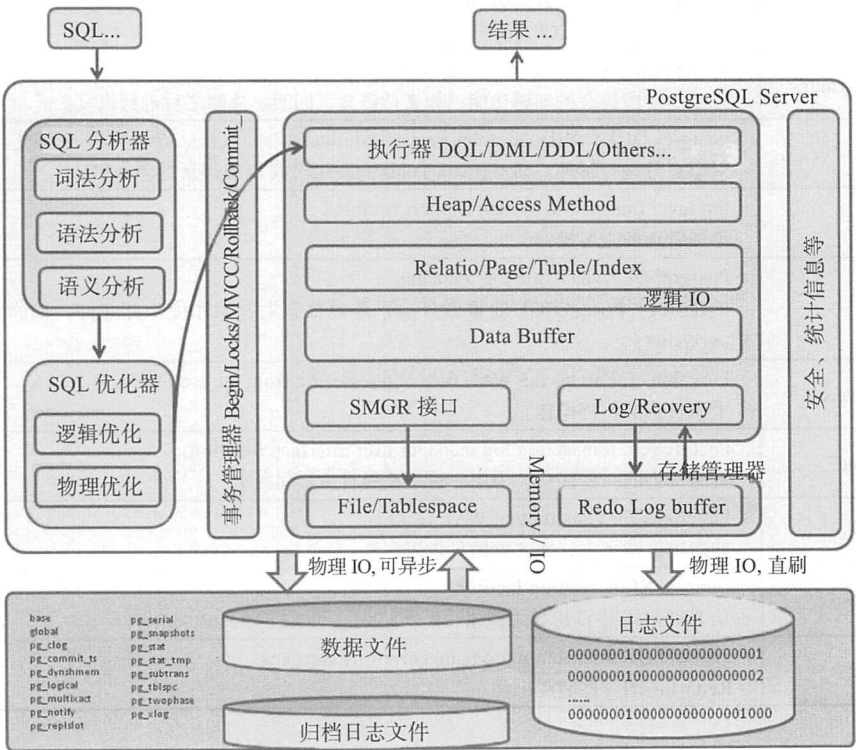


图 7-1 PostgreSQL 的基本架构图



- 上层是处于运行状态的结构，下层是外存的数据存储层。
- 上部分又可以分为四部分，左面的部分是分析器和优化器，主要完成 SQL 的分析和优化。第三部分是执行器，执行器要加载数据进入数据缓冲区，然后解析物理页面，数据被访问的形式是通过 heap 的相关操作完成；第二和第四部分，贯穿整个执行阶段，尤其是事务管理，在每一层都存在，而且是以事务锁的形式对数据库中的并发操作进行恰当的管理。

事务管理有一些基本的函数，这些函数分为不同的层次：

- **事务启动层**：由 start\_xact\_command() 和 finish\_xact\_command() 负责隐式地启动事务和结束事务。
- **事务状态识别层**：负责真正管理事务的启动、提交和回滚等操作。此过程虽然简单，但是需要识别事务的状态，在这一层判断事务的状态变迁是否合法，进而做出选择，所以是事务状态机的真正所在之处。所包括的函数如表 7-4 所示，通过三个函数，实现了事务的基本管理功能。
- **事务资源分配层**：内部真正的事务启动、清理等事务管理过程层。这一层，是事务真正的事务处理过程，实现了真正意义上的事务管理功能、嵌套事务处理等。所包括的函数如表 7-5 所示。

表 7-4 事务状态识别层事务处理函数表

函数名	功能
StartTransactionCommand	调用 StartTransaction() 函数，做真正的事务启动的处理
CommitTransactionCommand	调用 CommitTransaction () 函数，做真正的事务回滚的处理
AbortCurrentTransaction	调用 AbortTransaction ()、CleanupTransaction() 等函数，做真正的事务回滚的处理

表 7-5 内部事务处理函数表

函数名	功能
StartTransaction	真正的事务开始功能，如做资源的初始化、事务相关的内存的初始化等工作，详见 7.3.1 节
CommitTransaction	真正的事务提交功能，详见 7.3.2 节
AbortTransaction	真正的事务回滚功能，详见 7.3.4 节
CleanupTransaction	事务回滚的一部分，执行一些收尾工作，释放资源等
StartSubTransaction	开始子事务。如下四种操作，类似如上的四种事务相关操作。对比同类相似的事务操作，可以体会事务和子事务的差别，从而掌握嵌套事务的实现方式
CommitSubTransaction	提交子事务
AbortSubTransaction	回滚子事务
CleanupSubTransaction	清理子事务

- **面向用户的命令层**：直接接受用户发出的事务相关的命令由 standard\_ProcessUtility() 函数根据命令类型进行调度，作为事务状态机的一部分参与事务的管理工作。如表 7-6 中的函数，负责处理用户输入的事务操作类的 SQL 语句，如 BEGIN、COMMIT 等之类的

命令。但是，这些命令的执行，还是要经过前三层，即使用诸如“BEGIN”命令开启事务，其实也是处于上述隐式事务管理的一部分，示例参见 7.2.1 节的“例 1”。这四个层次，是事务管理器的接口，把事务处理的界限标识得非常清晰。

表 7-6 用户层事务处理函数表

函数名	功能
BeginTransactionBlock	对应 SQL 命令 BEGIN
EndTransactionBlock	对应 SQL 命令 COMMIT
UserAbortTransactionBlock	对应 SQL 命令 ROLLBACK
DefineSavepoint	对应 SQL 命令 SAVEPOINT
RollbackToSavepoint	对应 SQL 命令 ROLLBACK TO
ReleaseSavepoint	对应 SQL 命令 RELEASE

7.2 事务管理的基础

本节介绍 PostgreSQL 的事务管理的基础代码，主要涉及基本的数据结构如事务的状态和事务的数据结构。另外，粗略地讨论了与事务紧密相关 REDO 日志等相关内容。

7.2.1 事务状态

在 PostgreSQL 的源码中，经常可以看到“transaction block”这个词。

“transaction block”是一组 SQL 命令的集合，用以标识一个带有生命周期的“事务”。“transaction block states”则标识“有生命周期的一个事务块”在其生命周期中所处的阶段，每个阶段由不同的状态值标识，这就是 TBlockState 这个枚举类型的含义。

这个枚举类型实际上定义的是一个有限状态自动机的多个确定的、有限个数的状态。而在理解事务的过程中，需要根据这些个状态，去挖掘“动态的、状态互相转化的过程”。所以 TBlockState 是数据库内部用于实现事务管理的一个状态机标识，是标识状态间的转换的，不是用户可见、可理解的一个事务状态，本书把这种状态称为事务有限状态机的状态，或事务块的状态。

而用户可见的一个事务状态，使用 TransState 标识。

事务管理模块，就是依靠这两种状态，进行过程变迁，即状态转换的。

1. 事务块的状态

下面是 TBlockState 数据结构的定义：

```
/* transaction block states - transaction state of client queries //事务块的状态
*
* Note: the subtransaction states are used only for non-topmost transactions;
the others appear only in the topmost transaction. */
```



```

typedef enum TBlockState //枚举了事务块的所有状态，据此可以掌握一个session中带有ACID
                        特性的任务（即“事务”）被数据库引擎所重点关注的、其生命周期内的阶段有哪些
{
    /* not-in-transaction-block states */
    TBLOCK_DEFAULT, /* idle */ //没有事务块启动，就如同没有进程在CPU上运转，所以一个
                        session处于空闲状态（idle）
    TBLOCK_STARTED, /* running single-query transaction */ //简化的事务状态，标识单
    一SQL语句命令块（注意不只是SELECT命令）。可理解为单语句事务，可以参考IsTransactionBlock()函
    数体会其用途。一些操作如CLUSTER、ALTER DATABASE等命令需要在单语句事务的环境下运行，即不能
    其他语句同处于一个事务块内

    /* transaction block states */ //用以识别内部的事务状态，细致地细化了事务的过程，
    对于不同节点需要做出不同的处理
    TBLOCK_BEGIN,          /* starting transaction block */ //事务启动
    TBLOCK_INPROGRESS,     /* live transaction */ //事务进行中
    TBLOCK_PARALLEL_INPROGRESS, /* live transaction inside parallel worker */
    //正处于并行状态
    TBLOCK_END,            /* COMMIT received */ //事务结束，进行提交
    TBLOCK_ABORT,          /* failed xact, awaiting ROLLBACK */
    //事务被撤销（如报错准备结束事务），但回滚动作还没有开始
    TBLOCK_ABORT_END,      /* failed xact, ROLLBACK received */
    //事务被回滚，但回滚动作还没有完成，准备开始执行CleanupTransaction()
    TBLOCK_ABORT_PENDING,  /* live xact, ROLLBACK received */
    //用户发出回滚命令ROLLBACK
    TBLOCK_PREPARE,        /* live xact, PREPARE received */
    //“PREPARE TRANSACTION”，用于两阶段提交

    /* 子事务的状态，类似事务的状态，所以含义可以参考如上 */
    TBLOCK_SUBBEGIN,       /* starting a subtransaction */
    TBLOCK_SUBINPROGRESS,  /* live subtransaction */
    TBLOCK_SUBRELEASE,     /* RELEASE received */
    TBLOCK_SUBCOMMIT,      /* COMMIT received while TBLOCK_SUBINPROGRESS */
    TBLOCK_SUBABORT,       /* failed subxact, awaiting ROLLBACK */
    TBLOCK_SUBABORT_END,   /* failed subxact, ROLLBACK received */
    TBLOCK_SUBABORT_PENDING, /* live subxact, ROLLBACK received */
    TBLOCK_SUBRESTART,     /* live subxact, ROLLBACK TO received */
    TBLOCK_SUBABORT_RESTART /* failed subxact, ROLLBACK TO received */
} TBlockState;

```

## 2. 事务的状态

下面是 TransState 的定义，表示一个事务的状态：

```

typedef enum TransState
{
    TRANS_DEFAULT, /* idle */ //事务空闲，即没有事务在运行
    TRANS_START,   /* transaction starting */ //事务在启动中

```



```
TRANS_INPROGRESS,    /* inside a valid transaction */    //事务在运行中
TRANS_COMMIT,         /* commit in progress */    //事务提交
TRANS_ABORT,          /* abort in progress */    //事务回滚
TRANS_PREPARE         /* prepare in progress */    //事务在两阶段提交中的PREPARE阶段
} TransState;
```

3. 事务状态的转换

首先，我们来看几个示例，如表 7-7 所示。

表 7-7 事务状态转换示例说明表

	执行命令的前提条件	执行的命令
例 1	无	BEGIN
例 2	只执行过一条“BEGIN”命令	COMMIT
例 3	只执行过一条“BEGIN”命令	ROLLBACK
例 4	无	SELECT...
例 5	只执行过一条“BEGIN”命令	SELECT...
例 6	只执行过一条“BEGIN”和“SELECT”命令	SELECT... 一个不存在的表，让事务回滚

说明：

- ❑ 例 1 的目的，是观察一个事务开始时，函数间的调用关系以及事务相关状态的变迁关系；
- ❑ 例 1 是例 2 和例 3 的前提，在事务开始后，直接提交事务和回滚事务，则能看出事务“A”特性的 2 种不同结果（要么提交要么回滚）下事务相关状态的变迁关系；
- ❑ 例 1 和例 4 做一个对比，相似之处都是要“开始一个事务”，不同之处在于例 4 要开启一个隐含事务；
- ❑ 例 4 和例 5 做一个对比，相似之处都是要执行一个查询语句，不同之处在于例 5 先要开启一个显式事务，然后才执行 SELECT 命令；
- ❑ 例 6 是在例 5 基础上，执行一条错误的 SQL 语句，引发回滚操作，借以对比例 3。

例 1：执行 BEGIN 开始一个事务，函数调用关系与状态转换的关系如下：

```
exec_simple_query()
    start_xact_command()
        StartTransactionCommand()
            blockState = TBLOCK_DEFAULT //执行StartTransaction之前的状态
            StartTransaction()           //隐式开始事务
                state <= TRANS_START //“<=”表示赋值操作，如下相同
                一些初始化操作等
                state <= TRANS_INPROGRESS
                blockState <= TBLOCK_STARTED //执行StartTransaction之后，状态发生变化
            PortalRun()->...->standard_ProcessUtility() //对事务块中的每一条语句
            BeginTransactionBlock()
//真正开始执行“BEGIN”这个SQL命令，但是，请注意，这是在之前已经隐式开始事务之后才进行的
```

```

        blockState = TBLOCK_STARTED //初始状态
        blockState <= TBLOCK_BEGIN //由TBLOCK_STARTED转为TBLOCK_BEGIN
    finish_xact_command()
    CommitTransactionCommand()
        blockState = TBLOCK_BEGIN //初始状态
        blockState <= TBLOCK_INPROGRESS //由TBLOCK_BEGIN转为TBLOCK_INPROGRESS
        //执行“BEGIN”这个SQL命令结束之际

```

例2：执行 BEGIN 开始一个事务之后，直接执行 COMMIT 命令，函数调用关系与状态转换的关系如下：

```

exec_simple_query()
    start_xact_command()
        StartTransactionCommand()
            blockState = TRANS_INPROGRESS //此时，事务块的状态还是TRANS_INPROGRESS
        PortalRun()->...->standard_ProcessUtility() //对事务块中的每一条语句
        EndTransactionBlock() //真正开始执行“COMMIT”这个SQL命令
            blockState = TRANS_INPROGRESS //此时，事务块的状态还是TRANS_INPROGRESS
            blockState <= TBLOCK_END //由TBLOCK_END转为TBLOCK_END
    finish_xact_command()
        CommitTransactionCommand()
            blockState = TBLOCK_END
            CommitTransaction()
                state = TRANS_INPROGRESS
                一些预提交操作，如执行触发器、关闭大对象
                state = TRANS_COMMIT
                一些提交操作和释放资源的操作等，如调用RecordTransactionCommit()记录提交情况
                state <= TRANS_DEFAULT
            blockState <= TBLOCK_DEFAULT

```

例3：执行 BEGIN 开始一个事务之后，直接执行 ROLLBACK 命令，函数调用关系与状态转换的关系如下：

```

exec_simple_query()
    start_xact_command()
        StartTransactionCommand()
            blockState = TRANS_INPROGRESS //此时，事务块的状态还是TRANS_INPROGRESS
        PortalRun()->...->standard_ProcessUtility() //对事务块中的每一条语句
        UserAbortTransactionBlock() //用户“主动地”执行“ROLLBACK”这个SQL命令
            blockState = TRANS_INPROGRESS //此时，事务块的状态还是TRANS_INPROGRESS
            blockState <= TBLOCK_ABORT_PENDING //由TBLOCK_END转为TBLOCK_ABORT_PENDING
            //目的是通知CommitTransactionCommand()取消事务
    finish_xact_command()
        CommitTransactionCommand()
            blockState = TBLOCK_ABORT_PENDING
            AbortTransaction() //不管是主动还是被动，都需要回滚事务
                state = TRANS_INPROGRESS

```



```

        调用LWLockReleaseAll()释放锁等
        state = TRANS_ABORT
        调用AtEOXact_Parallel()清理并行操作和其他操作的环境和资源等
        CleanupTransaction()
        state <= TRANS_DEFAULT
        blockState <= TBLOCK_DEFAULT

```

例4：执行 SELECT 语句直接开始一个单语句事务，函数调用关系与状态转换的关系如下 (UPDATE 语句也相同)：

```

exec_simple_query()
    start_xact_command()
        StartTransactionCommand()
            blockState = TBLOCK_DEFAULT //执行StartTransaction之前的状态
            StartTransaction()           //单语句事务执行，直接开启一个事务
            state <= TRANS_START
            一些初始化操作等
            state <= TRANS_INPROGRESS
            blockState <= TBLOCK_STARTED //执行StartTransaction之后，状态发生变化
        finish_xact_command()
        CommitTransactionCommand()
            blockState = TBLOCK_STARTED //提交事务之前的状态
            CommitTransaction()          //单语句事务执行之后，如果不发生错误，直接提交
            state = TRANS_INPROGRESS
            一些预提交操作，如执行触发器、关闭大对象
            state = TRANS_COMMIT
            一些提交操作和释放资源的操作等，如调用RecordTransactionCommit()记录提交情况
            state <= TRANS_DEFAULT
            blockState <= TBLOCK_DEFAULT //提交事务之后设置新的状态

```

例5：执行 BEGIN 语句直接开始一个事务，然后执行 SELECT 语句，函数调用关系与状态转换的关系如下：

```

exec_simple_query()
    start_xact_command()
        StartTransactionCommand()
            blockState = TBLOCK_INPROGRESS //执行StartTransaction之前的状态
            //已经处于一个事务块中，所以不用调用StartTransaction()
        finish_xact_command()
        CommitTransactionCommand()
            blockState = TBLOCK_INPROGRESS //提交事务之前的状态
            CommandCounterIncrement()
            //SELECT命令执行结束，因事务没有结束，所以不需要调用CommitTransaction()

```

例6：在例5基础上，执行一条错误的 SQL 语句，引发回滚操作：

```

PostgresMain()
    exec_simple_query()

```





```

start_xact_command()
    StartTransactionCommand()
        blockState = TRANS_INPROGRESS //此时，事务块的状态还是TRANS_INPROGRESS
AbortCurrentTransaction() //事务管理器“被动地”执行回滚事务的操作
    AbortTransaction() //不管是主动还是被动，都需要回滚事务
        state = TRANS_INPROGRESS
        调用LWLockReleaseAll()释放锁等
        state <= TRANS_ABORT
        调用AtEOXact_Parallel()清理并行操作和其他操作的环境和资源等
    blockState <= TBLOCK_ABORT //标识事务块的状态是回滚

```

## 7.2.2 事务体

事务的状态结构体，是一个事务在数据库引擎执行的过程中，与周边环境交互的状况（如事务标识transactionId、是否处于恢复状态startedInRecovery、资源的属主是谁curTransactionOwner等）和对周边资源的使用状况（如curTransactionContext）等信息的记录体。

```

//transaction state structure
typedef struct TransactionStateData
{
    TransactionId    transactionId;    /* my XID, or Invalid if none */
//本事务的事务标志
    SubTransactionId subTransactionId; /* my subxact ID */    //本事务的子事务标志
    char             *name;            /* savepoint name, if any */
//一个子事务的保存点名称（如果存在，则是一个子事务）
    int              savepointLevel;   /* savepoint level */    //子事务的嵌套层次
    TransState       state;            /* low-level state */    //本事务的事务状态标志
    TBlockState      blockState;       /* high-level state */
//本事务的事务块状态机的状态标志
    int              nestingLevel;     /* transaction nesting depth */
//事务的嵌套层次
    int              gucNestLevel;     /* GUC context nesting depth */
//GUC，全局参数，用户可指定的参数。在不同的子事务层可以有子事务自己的配置参数
    MemoryContext     curTransactionContext; /* my xact-lifetime context */
//事务的内存上下文，事务可以使用的内存空间是一棵树，不同事务使用的内存空间就是树上的不同的叉
    ResourceOwner     curTransactionOwner; /* my query resources */
//事务使用的资源上下文，如数据缓冲区、临时文件句柄、锁等，对事务可用的资源进行管理
    TransactionId     *childXids;      /* subcommitted child XIDs, in XID order */
    int               nChildXids;      /* # of subcommitted child XIDs */
    int               maxChildXids;    /* allocated size of childXids[] */
    Oid               prevUser;        /* previous CurrentUserId setting */
//事务的用户
    int               prevSecContext;   /* previous SecurityRestrictionContext */
//事务的安全限制上下文
    bool              prevXactReadOnly; /* entry-time xact r/o state */

```





```
//事务读写属性
bool                startedInRecovery;    /* did we start in recovery? */
//是否是执行恢复操作的事务
bool                didLogXid;            /* has xid been included in WAL record? */
//标识本事务是否已经被记录到日志中
int                 parallelModeLevel;    /* Enter/ExitParallelMode counter */
//事务并行执行的计数器
struct TransactionStateData *parent;      /* back link to parent */
//子事务指向父事务的指针，构成链表
} TransactionStateData;
```

在 `xact.c` 文件中，定义了 `CurrentTransactionState` 如下，在需要标识、使用事务状态的时候，多使用此变量。

```
static TransactionState CurrentTransactionState = &TopTransactionStateData;
```

### 7.2.3 事务运行的简略过程

在 PostgreSQL 的内部，一个事务的执行的简略过程如下，粗略地表明了事务启动到结束的上下文状况。

```
static void
exec_simple_query(const char *query_string) //执行入口参数指定的SQL语句
{...
    /* Start up a transaction command. All queries generated by the query_string
    will be in this same command block,
    * *unless* we find a BEGIN/COMMIT/ABORT statement; we have to force a new
    xact command after one of those,
    * else bad things will happen in xact.c. (Note that this will normally
    change current memory context.) */
    start_xact_command(); //如果事务没有启动，则启动一个事务
    ...
    // Do basic parsing of the query or queries (this should be safe even if we
    are in aborted transaction state!)
    parsetree_list = pg_parse_query(query_string); //对SQL语句进行语法分析、查询优化等
    ...
    // Run through the raw parsetree(s) and process each one.
    foreach(parsetree_item, parsetree_list) //遍历输入的所有的SQL语句
    {...
        /* Get the command name for use in status display (it also becomes the
        default completion tag, down inside PortalRun).
        * Set ps_status and do any special start-of-SQL-command processing
        needed by the destination */
        commandTag = CreateCommandTag(parsetree);
        set_ps_display(commandTag, false);
        BeginCommand(commandTag, dest); //目前什么也不执行，一个空壳函数
```



```

...
    /* Make sure we are in a transaction command */
    start_xact_command(); //如果事务没有启动, 则启动一个事务

...
    /* We don't have to copy anything into the portal, because everything we
       are passing here is in MessageContext,
       * which will outlive the portal anyway. */
    PortalDefineQuery(portal,...);

...
    // Run the portal to completion, and then drop it (and the receiver).
    (void) PortalRun(portal,...);
//SQL语句按照执行计划执行的过程, 事务管理器在此过程中发挥巨大作用

...
    if (IsA(parsetree, TransactionStmt))
    {
        //If this was a transaction control statement, commit it. We will
        start a new xact command for the next command (if any).
        finish_xact_command();
//如果遇到一个事务控制语句, 则提交之前执行过的内容, 这样则可以开始一个新事务
    }
    else if ...

        // Tell client that we're done with this query.
        EndCommand(completionTag, dest);
    }/* end loop over parsetrees */

    // Close down transaction statement, if one is open.
    finish_xact_command(); //如果需要结束事务, 则结束

...
}

```

## 7.3 事务操作

在 PostgreSQL 事务管理模块内部, 正式启动一个事务, 是从 `start_xact_command()` 函数开始的。真正开始一个事务, 是从 `StartTransactionCommand()` 函数开始的 (事务状态机的起始处)。而真正执行一个事务, 却是从 `StartTransaction()` 函数开始的。

事务相关的其他操作, 与此相似, 详情可参见 7.1.2 节对于事务层次的分析。

### 7.3.1 开始事务

本节讲述 `StartTransaction()` 函数的实现。





## 1. 开始事务的函数

事务开始，需要为事务的执行准备初始的环境，`StartTransaction()` 函数就是做事务启动的准备工作。

```
static void
StartTransaction(void)
//开始一个事务，需要为事务准备执行环境，包括内存、资源、一些变量的初始值等
{...
    CurrentTransactionState = s;
    // CurrentTransactionState是查询事务的状态的重要变量，标识当前的事务的状况
    ...

    //set the current transaction state information appropriately during start processing
    s->state = TRANS_START;           //设置事务状态
    s->transactionId = InvalidTransactionId;    /* until assigned */
    //事务开始的时候，没有事务号

    if (RecoveryInProgress()) //处于恢复状态的事务，事务属性应是只读的
    {
        s->startedInRecovery = true;
        XactReadOnly = true;
    }
    else
    {
        s->startedInRecovery = false;
        XactReadOnly = DefaultXactReadOnly;
    }
    XactDeferrable = DefaultXactDeferrable; //赋予初始值
    XactIsoLevel = DefaultXactIsoLevel;
    forceSyncCommit = false;
    MyXactAccessedTempRel = false;
    ...

    // must initialize resource-management stuff first
    AtStart_Memory();           //初始化事务可以使用的内存
    AtStart_ResourceOwner();    //初始化事务可以使用的资源
    ...

    xactStartTimestamp = stmtStartTimestamp; //赋予初始值
    xactStopTimestamp = 0;
    pgstat_report_xact_timestamp(xactStartTimestamp);
    s->nestingLevel = 1;           //赋予初始值
    s->gucNestLevel = 1;
    s->childXids = NULL;
    s->nChildXids = 0;
    s->maxChildXids = 0;
    ...

    //initialize other subsystems for new transaction
```



```
AtStart_GUC(); //初始化事务可以使用的GUC参数，每个子事务可以拥有自己的参数值
AtStart_Cache(); //初始化事务需要知晓的全局元信息失效的cache (the system catalog caches)
AfterTriggerBeginXact(); //初始化触发器相关环境
s->state = TRANS_INPROGRESS; //事务启动完成，接下来事务进入运行状态

ShowTransactionState("StartTransaction");
}
```

PostgreSQL 的事务启动工作，和 InnoDB 的事务启动工作，有所不同，可以比较 10.3.2 节。PostgreSQL 的事务管理模块化痕迹很重，可以明显看到对于事务要使用的内存、资源等提前进行了谋划。而 InnoDB 事务的启动工作，是复用事务池中的事务对象，然后给事务对象的属性赋予初始值。两者在给事务对象的属性赋予初始值方面基本一致，但缺少对内存系统地进行管理的策略。InnoDB 较 PostgreSQL 所多做的一点是为事务分配了回滚段，在回滚段这一特性上，PostgreSQL 缺乏没有实现。

2. 获取事务 ID

执行一个 SQL 语句，在语法解析和查询优化阶段，事务管理器是不介入的，获得查询执行计划之后，执行器开始介入，才可能进行事务 ID 的分配。普通的查询语句，不占用宝贵的事务号，即不分配事务 ID（InnoDB 对于只读事务，也不分配事务 ID）。而其他的 SQL，需要涉及锁操作有修改意图的，需要分配事务 ID。如表 7-8 所示，左侧是一个带有更新意愿的查询语句，右侧是一个更新语句，都需要分配事务 ID，分配事务 ID 的调用栈如表 7-8 所示。

表 7-8 事务 ID 分配调用栈示例表

SELECT * FROM t1 FOR UPDATE;	UPDATE t1 SET c=222 WHERE a=2;
exec_simple_query	exec_simple_query
PortalRun	PortalRun
PortalRunSelect	PortalRunSelect
ExecutorRun // 开始执行	ExecutorRun // 开始执行
standard_ExecutorRun	standard_ExecutorRun
ExecutePlan	ExecutePlan
ExecProcNode	ExecProcNode
ExecLockRows	ExecModifyTable
heap_lock_tuple // 给元组加锁	ExecUpdate
GetCurrentTransactionId	heap_update // 更新元组，首先获得事务的 ID
AssignTransactionId // 分配事务 ID	GetCurrentTransactionId
	AssignTransactionId // 分配事务 ID

如图 7-2 所示的 AssignTransactionId() 函数的调用关系，可以清楚地看到，DML 类的操作需要分配事务的 ID。除此之外的很多操作，都需要获取当前事务的 ID，更为详细的信息，请读者参阅代码。



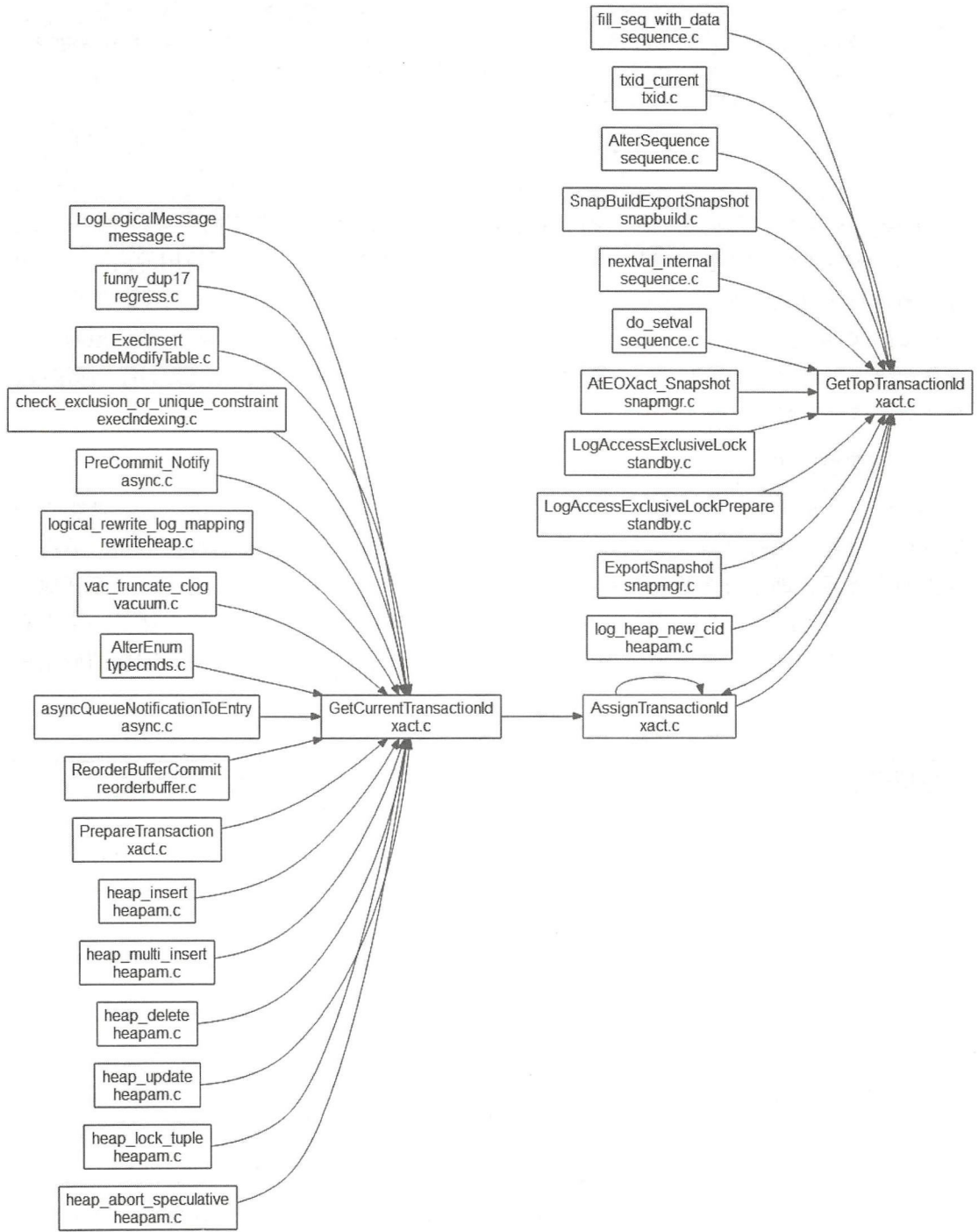


图 7-2 分配事务 ID (AssignTransactionId()) 函数的调用关系图





## 7.3.2 事务提交

事务的提交操作，包括三个关键步骤：

- 提交标志的设置：执行 “s->state = TRANS\_COMMIT” 设置事务的状态。
- 记载到日志：调用 RecordTransactionCommit() 记录日志并刷出日志（不一定刷出到物理存储，需要根据 fsync 参数的设置值确定）。
- 释放锁：释放锁的位置是在设置提交标志之前还是之后，可以表明使用的封锁算法是否是 SS2PL。具体差异如图 2-5 所示，请参阅 2.3.2 节。

```
CommitTransaction(void) //执行事务提交时，事务收尾工作
{...
    //执行被延迟的触发器
    ...
    if (IsInParallelMode())
        AtEOXact_Parallel(true); //清理并行相关内容
    AfterTriggerEndXact(true); /* Shut down the deferred-trigger manager */
    PreCommit_on_commit_actions(); //为临时表处理“ON COMMIT”操作

    /* close large objects before lower-level cleanup */
    AtEOXact_LargeObject(true); //关闭大对象，大对象在PG中被拆分为多个小的元组存储

    PreCommit_CheckForSerializationFailure(); //事务提交前，对可串行化事务进行合法性检
    查，详情参见9.4.5节
    PreCommit_Notify(); // Insert notifications sent by NOTIFY commands into the queue.
    HOLD_INTERRUPTS(); /* Prevent cancel/die interrupt while cleaning up */
    //执行到此处，不能被撤销，只能继续向下执行

    /* Commit updates to the relation map --- do this as late as possible */
    AtEOXact_RelationMap(true); //调用perform_relmap_update()->write_relmap_file()
    更新系统关系表，更新前，需要把恢复信息写入日志以备不测
    s->state = TRANS_COMMIT; //设置为提交状态，这是事务提交的标志。之后，才能释放锁等
    s->parallelModeLevel = 0;

    if (!is_parallel_worker)
    {
        //We need to mark our XIDs as committed in pg_clog. This is where we durably commit.
        latestXid = RecordTransactionCommit(); //在clog中记载事务提交这个事实
    }
    else
    {...} //处理并行的情况
    TRACE_POSTGRES_SQL_TRANSACTION_COMMIT(MyProc->lxid);
    ProcArrayEndTransaction(MyProc, latestXid); //通知其他进程，本进程没有事务在执行了

    /* The ordering of operations is not entirely random. The idea is:
    //注意清理的顺序如下
```





```
* release resources visible to other backends (eg, files, buffer pins);
then release locks; then release backend-local resources.
```

```
* We want to release locks at the point where any backend waiting for us
will see
```

```
* our transaction as being fully cleaned up. */
```

**CallXactCallbacks**(is\_parallel\_worker ? XACT\_EVENT\_PARALLEL\_COMMIT : XACT\_EVENT\_COMMIT); //提交操作之后,通过钩子函数,执行的清理工作。目前主要用于外部表和标记安全相关的操作。如果不涉及外部表和标记安全这两个特性,本地事务则不需要执行此回调函数规定的清理顺序,所以特别注意不要和本地事务的清理工作相混淆(一个重要的混淆点,在于锁的释放是在事务标识为提交之后才进行,则误以为PostgreSQL使用了两阶段封锁协议,此处实则对于外部表和标记安全相关的操作而言的。而PostgreSQL自身的并发管理是使用的SS2PL和MVCC,记录锁的施加是在用后即释放的,如在heap\_update()函数中调用heap\_acquire\_tuplock()->LockTuple()加锁,然后在这个函数内调用UnlockTuple()释放掉锁;而DDL类的锁确实要被释放掉,只是稍候释放)

```
ResourceOwnerRelease(TopTransactionResourceOwner, RESOURCE_RELEASE_
BEFORE_LOCKS, true, true); //资源的释放
```

```
AtEOXact_Buffers(true); /* Check we've released all buffer pins */
```

```
AtEOXact_RelationCache(true); /* Clean up the relation cache */
```

```
AtEOXact_Inval(true); // Make catalog changes visible to all backends.
```

```
AtEOXact_MultiXact();
```

//调用**ProcReleaseLocks()**->**LockReleaseAll()**释放锁,调用**ReleasePredicateLocks()**释放谓词锁(SS1技术实现序列化,参见第9章)。从此处看,PostgreSQL使用了SS2PL技术,在提交点之后才释放锁。但是,需要注意的是,这里释放的都是DDL类型的事务锁,而不包括DML类型的事务锁。所以PostgreSQL使用的SS2PL技术只是针对DDL类操作,而第9章讲述的MVCC则是DML操作并发控制技术。例如,更新操作的元组锁的释放是在update操作执行函数heap\_update()结束前就释放的,参见8.3.3节“3.行级锁的加锁和释放”。

```
ResourceOwnerRelease(TopTransactionResourceOwner,
```

```
RESOURCE_RELEASE_LOCKS, true, true);
```

ResourceOwnerRelease(TopTransactionResourceOwner, RESOURCE\_RELEASE\_AFTER\_LOCKS, true, true); //依序释放资源如catcache、plancache、snapshot、临时文件、索引扫描清理等

```
smgrDoPendingDeletes(true); //事务提交,删除relation,清理物理存储层
```

```
AtEOXact_CatCache(true); /* Check we've released all catcache entries */
```

```
AtCommit_Notify(); //事务提交,清理内存
```

```
AtEOXact_GUC(true, 1); //事务提交,清理GUC变量等
```

```
...
```

```
AtEOXact_Files(); //事务提交,清理文件
```

```
...
```

```
AtEOXact_PgStat(true); //事务提交,清理统计
```

```
AtEOXact_Snapshot(true); //事务提交,清理快照
```

```
pgstat_report_xact_timestamp(0);
```

```
...
```

```
AtCommit_Memory(); //事务提交,清理内存
```

```
s->transactionId = InvalidTransactionId; //事务的属性回归初始值
```

```
...
```



```
s->state = TRANS_DEFAULT;           //事务提交完成

RESUME_INTERRUPTS();

}
```

### 7.3.3 日志落盘

在事务提交的时候，日志要提前于数据刷出到外存，这样的技术称为“预先日志”，即WAL。目的是保证系统故障或介质故障发生时，事务符合“原子性”和数据的“一致性”：

- 以日志记载了的时间点为“COMMIT\_TS\_SETTS”日志提交的标志，在这之前，尽管一个事务块内记载了很多日志，但在记载时间点为“COMMIT\_TS\_SETTS”日志提交的标志之前系统崩溃，则事务不被重新执行，相当于事务没有提交，保证了系统崩溃前的事务因未提交而呈现原子特性，数据也是旧数据，保持了一致性。
- 以日志记载了的时间点为“COMMIT\_TS\_SETTS”日志提交的标志，在这之后，数据被刷出之前系统崩溃，则事务会被重新执行，然后事务能够正确提交，保证了“事务是可以正确提交的”，所以保证了事务的原子性，即也保证了数据的一致性。

所以，日志一定要在提交事务前刷出完成。RecordTransactionCommit() 函数就是用于刷出日志的。

```
static TransactionId
RecordTransactionCommit(void)
{...
    if (!markXidCommitted) //事务ID不合法
    {...}
    else
    {...
        START_CRIT_SECTION();
        MyPgXact->delayChkpt = true;
        SetCurrentTransactionStopTimestamp();

        XactLogCommitRecord(xactStopTimestamp, nchildren, children, nrels, rels,
nmsgs, invalMessages, //记录提交信息到日志
                           RelcacheInitFileInval, forceSyncCommit,
InvalidTransactionId /* plain commit */ );
        ...

        //记录提交时间点的信息到日志（调用XLogInsert(RM_COMMIT_TS_ID, COMMIT_TS_SETTS)
把时间点标志信息写入日志），并记载clog
        TransactionTreeSetCommitTsData(xid, nchildren, children, replorigin_
session_origin_timestamp, replorigin_session_origin, false);
    }
    if ((wrote_xlog && markXidCommitted && synchronous_commit > SYNCHRONOUS_
COMMIT_OFF) || forceSyncCommit || nrels > 0)
    {
```





```

        XLogFlush(XactLastRecEnd); //刷出WAL日志
    if (markXidCommitted)
        TransactionIdCommitTree(xid, nchildren, children);
        //更新clog, 把事务完成的信息写入clog
    }
    else
    {...} //异步提交
...
    /* Compute latestXid while we have the child XIDs handy */
    latestXid = TransactionIdLatest(xid, nchildren, children);
    //注意返回值latestXid在上层调用函数中的用法
...
    return latestXid;
}

```

那么, 哪一个动作代表事务的提交呢? 是 `CommitTransaction()` 函数中设置了事务的状态, 还是 `RecordTransactionCommit()` 函数中刷出了日志? 读者可以仔细思考一下这个问题。7.3.5 节将进一步分析这个问题。

另外, `XactLogCommitRecord()` 函数表示了 REDO 日志的提交, 在这个函数中, 把日志记录的标志, 设置为了 `XLOG_XACT_COMMIT`, 表示这是一个事务的日志提交记录。在系统恢复的过程中, 调用 `recoveryStopsBefore()` 函数根据事务的日志提交标志 `XLOG_XACT_COMMIT` 设定恢复的结束点 (以事务为恢复单位, 保证系统因崩溃而恢复后的事务原子性)。

## 7.3.4 事务回滚

事务回滚会撤销事务中发生的各种修改操作, 而一个正常的回滚操作 (发出 `ROLLBACK` 命令), 需要通过调用 `AbortTransaction()`、`CleanupTransaction()` 这两个函数完成事务的回滚任务。

### 1. 事务回滚操作

```

static void
AbortTransaction(void)
{...
    /* Make sure we have a valid memory context and resource owner */
    AtAbort_Memory();           //切换事务所使用的内存上下文
    AtAbort_ResourceOwner();    //切换资源属主

    /* Release any LW locks we might be holding as quickly as possible.
     * (Regular locks, however, must be held till we finish aborting.)
     * Releasing LW locks is critical since we might try to grab them again while cleaning up!
     */
    LWLockReleaseAll();         //释放轻量锁, 这是一种系统锁, 非事务类型的锁, 参见8.2.2节
}

```





```

/* Clear wait information and command progress indicator */
pgstat_report_wait_end();
pgstat_progress_end_command();

/* Clean up buffer I/O and buffer context locks, too */
AbortBufferIO();
UnlockBuffers();

XLogResetInsertion(); /* Reset WAL record construction state */

/* Also clean up any open wait for lock, since the lock manager will choke
 * if we try to wait for another lock before doing this. */
LockErrorCleanup(); //锁的清理, 如本事务所申请的锁正在等待其他事务释放, 则需要把本事务
所申请的锁从等待队列中删除

/* If any timeout events are still active, make sure the timeout interrupt is scheduled.
 * This covers possible loss of a timeout interrupt due to longjmp'ing out of
the SIGINT handler (see notes in handle_sig_alarm).
 * We delay this till after LockErrorCleanup so that we don't uselessly
reschedule lock or deadlock check timeouts. */
reschedule_timeouts(); //清理超时机制所遗留的

...
//set the current transaction state information appropriately during the
abort processing
s->state = TRANS_ABORT; //设置事务的回滚状态, 表明回滚操作已经生效

...
//do abort processing, 如下一些工作类似于事务提交阶段所做的环节, 只是每类操作侧重于
“abort”
AfterTriggerEndXact(false); /* 'false' means it's abort */
AtAbort_Portals();
AtEOXact_LargeObject(false);
AtAbort_Notify();
AtEOXact_RelationMap(false);
AtAbort_TwoPhase();
if (!is_parallel_worker)
    latestXid = RecordTransactionAbort(false); //注意回滚操作要调用
XactLogAbortRecord() 记录到REDO日志中, 然后调用TransactionIdAbortTree()
在clog中标识事务撤销
else
{
    ...
    XLogSetAsyncXactLSN(XactLastRecEnd); //并行方式的处理
}

...
if (TopTransactionResourceOwner != NULL)
//如果是顶层事务, 最重要的一个步骤, 是释放锁的过程
{
    ...

```





```

        ResourceOwnerRelease(TopTransactionResourceOwner,
        RESOURCE_RELEASE_BEFORE_LOCKS, false, true);
        AtEOXact_Buffers(false);    //清理缓存区
    ...
        ResourceOwnerRelease(TopTransactionResourceOwner, RESOURCE_RELEASE_LOCKS,
        false, true); //释放锁
        ResourceOwnerRelease(TopTransactionResourceOwner, RESOURCE_RELEASE_AFTER_
        LOCKS, false, true);
    ...
        AtEOXact_GUC(false, 1);    //清理GUC参数
    ...
    }
    ...
}

```

## 2. 事务清理操作

事务的清理操作，是事务已经完成了回滚过程中的主要回滚任务，然后需要对一些变量做清理的工作。

```

static void
CleanupTransaction(void)
{...
    AtCleanup_Portals();                /* now safe to release portal memory */
    AtEOXact_Snapshot(false);           /* and release the transaction's snapshots */

    CurrentResourceOwner = NULL;        /* and resource owner */
    if (TopTransactionResourceOwner)
        ResourceOwnerDelete(TopTransactionResourceOwner);
    s->curTransactionOwner = NULL;
    CurTransactionResourceOwner = NULL;
    TopTransactionResourceOwner = NULL;

    AtCleanup_Memory();                  //清理事务过程中所使用的内存，如把分配过的内存回收

    s->transactionId = InvalidTransactionId;
    s->subTransactionId = InvalidSubTransactionId;
    s->nestingLevel = 0;                  //重置全局的一些变量
    s->gucNestLevel = 0;
    s->childXids = NULL;
    s->nChildXids = 0;
    s->maxChildXids = 0;
    s->parallelModeLevel = 0;

    ...
    s->state = TRANS_DEFAULT;            //标识回滚操作完成
}

```





### 3. 隐式的事务回滚

AbortTransaction()、CleanupTransaction() 这两个函数一起完成回滚任务，之所以分开为两个函数，是因为 PostgreSQL 提供的事务实现机制依赖于 PostgreSQL 异常处理的实现方式。

PostgreSQL 提供了如下三个宏，用以实现异常的捕捉。

```
#define PG_TRY() \
do { \
    sigjmp_buf *save_exception_stack = PG_exception_stack; \
    //PG_exception_stack是一个进程级的全局变量，用以保存异常发生时的栈地址
    ErrorContextCallback *save_context_stack = error_context_stack; \
    //定义错误上下文
    sigjmp_buf local_sigjmp_buf; \
    //定义一个当前代码位置的标签
    if (sigsetjmp(local_sigjmp_buf, 0) == 0) \
    //参数local_sigjmp_buf，保存目前堆栈环境，然后将目前的地址作一个记号
    { \
        PG_exception_stack = &local_sigjmp_buf

#define PG_CATCH() \
    \
    //捕获异常
    } \
    else \
    //sigsetjmp()值为非零，则由siglongjmp()跳转回来
    { \
        PG_exception_stack = save_exception_stack; \
        //捕获异常的时候，恢复之前的异常栈
        error_context_stack = save_context_stack
        //捕获异常的时候，恢复之前的错误上下文

#define PG_END_TRY() \
    } \
    PG_exception_stack = save_exception_stack; \
    //捕获异常的时候，恢复之前的异常栈
    error_context_stack = save_context_stack; \
    //捕获异常的时候，恢复之前的错误上下文
    } while (0)
```

上述宏需要配对使用，构成一个完整的 TRY...CATCH 块，且如果继续向上层传递异常信息时，可以通过 PG\_RE\_THROW() 再次抛出异常。例如，PortalRun() 函数典型地使用了这样的用法：

```
PortalRun(...) //执行SQL语句
{...
    PG_TRY(); //代码置于TRY...CATCH块中，可以捕获代码中抛出的错误，如使用
    "ereport(ERROR,...)" 报告错误
    {...
        switch (portal->strategy)
        {
```



```

        case PORTAL_ONE_SELECT: //如查询等的执行
        case PORTAL_ONE_RETURNING:
        case PORTAL_ONE_MOD_WITH:
        case PORTAL_UTIL_SELECT:

...
            nprocessed = PortalRunSelect(portal, true, count, dest);

...
            result = portal->atEnd;
            break;
        case PORTAL_MULTI_QUERY: //如插入等的执行
            PortalRunMulti(portal, isTopLevel, dest, altdest, completionTag);

...
            break;
        default:
            elog(ERROR, "unrecognized portal strategy: %d", (int) portal->strategy);
            result = false; /* keep compiler quiet */
            break;
    }
}
PG_CATCH(); //代码置于TRY...CATCH块中
{
...
    //捕获错误, 做出处理
    PG_RE_THROW(); //然后, 继续向上层抛出错误。 执行 “if (PG_exception_stack !=
NULL) siglongjmp(*PG_exception_stack, 1);”
}
PG_END_TRY(); //代码置于TRY...CATCH块中
...
}

```

在一个会话进程的主函数 `PostgresMain()` 中, 实现由如下代码, 用以接受抛出的错误, 然后回滚事务。在这段代码中, 事务的回滚不是由用户发出回滚命令推动的, 而是事务执行的过程中检测到错误而主动通过 “`ereport(ERROR,...`” 报告错误而回滚事务 (如发现主键冲突: `ERROR: duplicate key value violates unique constraint "t2_pkey"`), 这种方式, 就是隐式回滚。此时, 需要通过 `AbortTransaction()` 复原一些事务处理过程中被改变了的环境、变量等; 之后, 还需要进入到事务有限状态自动机中 (`AbortCurrentTransaction()`) 继续事务的清理工作 (因为事务未必结束, 在一个事务块内发生了异常, 还需要用户自己手动发出回滚命令, 这时, 就直接执行 `CleanupTransaction()` 函数即可), 所以把事务回滚的操作拆分为两个函数应对不同阶段处理的需求。

```

PostgresMain(int argc, char *argv[], const char *dbname, const char *username)
{...
    if (sigsetjmp(local_sigjmp_buf, 1) != 0) //返回非0, 表示由siglongjmp()跳转回来,
    即从 “PG_RE_THROW()” 执行后返回到这里
    {...

```





```

/* Make sure libpq is in a good state */
pq_comm_reset();    //错误发生后，执行一些恢复环境的工作
EmitErrorReport();  /* Report the error to the client and/or server log */
...

AbortCurrentTransaction();
//回滚当前报告错误的事务。此处是隐式回滚事务的重要代码，当捕获错误，则回滚事务
...
}
...}

```

隐式的错误回滚机制，被 PostgreSQL 大量使用。例如 9.4.5 节谓词冲突检测需要回滚事务，就是通过报告错误而回滚事务的，不是主动调用回滚函数实现回滚。掌握隐式回滚这一点，对于通过事务回滚的方式理解读写依赖冲突处理的方式很有帮助。

### 7.3.5 clog

PostgreSQL 使用 clog 文件，记录日志的提交会回滚情况，主要的函数如下。

- TransactionIdDidCommit(): 判断事务是否已经提交。用于元组的可见性判断。
  - TransactionIdDidAbort(): 判断事务是否已经回滚。
  - TransactionIdCommitTree(): 在 clog 中设置给定的事务和子事务的状态为提交。
- 这些函数，依赖几个和事务的状态有关的宏，定义如下：

```

#define TRANSACTION_STATUS_IN_PROGRESS    0x00    //表示事务正在运行
#define TRANSACTION_STATUS_COMMITTED      0x01    //表示事务提交
#define TRANSACTION_STATUS_ABORTED        0x02    //表示事务回滚
#define TRANSACTION_STATUS_SUB_COMMITTED  0x03    //表示子事务提交

```

如果需要判断事务是否已经提交，就要依赖上述的宏，如下的函数是在元组可见性判断时，y 用于判断事务的状态：

```

bool    /* true if given transaction committed */    //用于判断事务是否已经提交
TransactionIdDidCommit(TransactionId transactionId)
{...
    xidstatus = TransactionLogFetch(transactionId);    //根据事务ID获取本事务的状态
    if (xidstatus == TRANSACTION_STATUS_COMMITTED)
//如果在clog里面记录了事务ID的状态为提交，则表示事务已经提交
        return true;
    ...
    return false;    // It's not committed.
}

```

TransactionIdDidCommit() 函数，被 HeapTupleSatisfiesMVCC() 等函数用来确认某个事务是否已经提交。所以，事务是否提交的标志，就是在 clog 中是否把事务的状态设置为 “TRANSACTION\_STATUS\_COMMITTED”。这个操作是发生在写日志之后的，所以 PostgreSQL 的日志技术是预先日志技术。





## 7.4 子事务的管理

在 PostgreSQL 代码内部, 有“子事务”相关的函数和源码文件, 如表 7-2 中的 `StartSubTransaction()` 等, 文件如“`subtrans.c`”中的描述也似子事务。但是, PostgreSQL 的 SQL 语法事务相关部分定义<sup>①</sup>, 却没有子事务的内容。而我们知道 PostgreSQL 支持“SAVEPOINT”即保存点, 那么 PostgreSQL 到底有没有支持子事务呢?

Wiki 对于保存点<sup>②</sup>的定义如下:

A savepoint is a way of implementing subtransactions (also known as **nested transactions**) within a relational database management system by indicating a point within a transaction that can be "rolled back to" without affecting any work done in the transaction before the savepoint was created.

这是在说, 保存点是一种实现嵌套类型的子事务的实现方法。所以 PostgreSQL 是通过保存点技术来实现嵌套子事务的。

### 7.4.1 子事务与父事务的区别

如表 7-2 所示, 子事务的管理, 同事务的管理一样, 包括四个函数, 分别实现子事务的初始化、提交、回滚、清理操作。这些基本类似于事务的管理。

而子事务的实现, 是嵌套在事务管理中的, 所以事务的初始化为子事务的执行提供了环境基础, 事务的处理机制适用于子事务的处理机制, 二者基本上是相似的。如下是子事务的初始化代码, 可以看到, 子事务也是在初始化一些内存空间和资源, 建立自己的通知机制等。但事务中的很多其他类型的初始化工作, 子事务不再重复, 而是依赖于父事务。

```
static void
StartSubTransaction(void)
{...
    s->state = TRANS_START;           //子事务同样以TRANS_START为子事务开始的状态标识
    AtSubStart_Memory();               //子事务的内存空间初始化
    AtSubStart_ResourceOwner();        //子事务的资源初始化
    AtSubStart_Notify();
    AfterTriggerBeginSubXact();
    s->state = TRANS_INPROGRESS;
    ...
}
```

但是每个子事务都有自己的不同的内存空间, 例如子事务和父事务的内存空间初始化就有不同。

```
static void
```

① 如9.6版本的文档: <https://www.postgresql.org/docs/9.6/static/sql-begin.html>

② <https://en.wikipedia.org/wiki/Savepoint>



```

AtSubStart_Memory(void) //子事务的内存空间初始化，是以当前事务的内存空间为根，
    然后在当前事务的内存空间之下分配各自的空间
{...
    CurTransactionContext = AllocSetContextCreate(CurTransactionContext,
        //以当前事务的内存空间为根
        "CurTransactionContext", ALLOCSET_DEFAULT_MINSIZE, ALLOCSET_DEFAULT_
        INITSIZE, ALLOCSET_DEFAULT_MAXSIZE);
    ...
}

static void
AtStart_Memory(void) //父事务的内存空间初始化，是以顶层的内存空间为根，然后在顶层的内存
    空间之下分配各自的空间
{...
    if (TransactionAbortContext == NULL)
        //以顶层的内存空间为根，分配所有子事务都需要的事务回滚内存上下文
        TransactionAbortContext = AllocSetContextCreate(TopMemoryContext,
            "TransactionAbortContext", 32 * 1024, 32 * 1024, 32 * 1024);

    TopTransactionContext =
        AllocSetContextCreate(TopMemoryContext,
            //以顶层的内存空间为根，分配所有父事务的内存上下文
            "TopTransactionContext", ALLOCSET_DEFAULT_MINSIZE,
            ALLOCSET_DEFAULT_INITSIZE, ALLOCSET_DEFAULT_MAXSIZE);
    ...
}

```

子事务的提交、回滚，也类似于父事务的提交、回滚，只是在其自己的空间内进行。具体实现上的差别，请参阅代码，在此不再赘述。

## 7.4.2 保存点

保存点的实现方式，是通过支持定义保存点名称，然后在数据库引擎内部开启一个子事务块（`PushTransaction()` 函数），之后进入事务有限状态自动机中调用 `StartSubTransaction()` 函数来初始化子事务，之后通过 `CommitSubTransaction()` 或 `AbortSubTransaction()` 来对子事务进行提交或回滚。

如果定义多个保存点，则在同一个顶层事务上下文“`TopTransactionContext`”上可以并列分配多个子事务的事务块的状态（`TransactionState`）。保存点的定义，可以并列也可以嵌套。

保存点的提交或回滚，则是以一个子事务块为单位，局部进行提交或回滚，但最终受父事务提交或回滚的影响。在保存点的提交或回滚时，一个子事务块内施加的 DDL 类型的锁，会被释放，不是等到父事务的提交或回滚时才进行释放。

保存点存在的意义：一是适当利用可以提高事务的并发度，这是因为子事务的提交或回滚是可以释放锁资源的；二是可以实现更加灵活的用户操作。



可以通过关注 `PushTransaction()` 和 `PopTransaction()` 函数、以及他们的调用上下文，来掌握保存点的实现，进而掌握 PostgreSQL 的子事务。

需要注意的一点，是 PostgreSQL 的保存点的实现，和 MySQL 的 InnoDB 的保存点的实现技术不同，PostgreSQL 采用事务块和子事务块，实现了保存点；而 InnoDB 则是通过在回滚段上遍历旧的操作而实现子事务的提交和回滚的（参见 10.3.8 节）。从代码可以看出，PostgreSQL 对保存点的实现更加统一、规整，而 InnoDB 则很巧妙地利用了回滚段实现了保存点。

## 7.5 本章小结

本章从 PostgreSQL 的事务管理模块讲起，首先简略概述了 PostgreSQL 事务处理相关的文件和功能，然后从数据库引擎整体讲述 PostgreSQL 的架构和事务在其中的体系层次。

在 7.2 节提出事务有限状态自动机，PostgreSQL 的事务管理就是在实现这么一个有限状态自动机。后续接着讲述了事务操作和子事务相关的操作。待到从子事务引出保存点的时候，实际上能够明白 PostgreSQL 的事务管理模型是支持嵌套子事务的。

对于自治事务，很明确，PostgreSQL 不支持。但是相关的一个讨论，可以参考：

[https://wiki.postgresql.org/wiki/Autonomous\\_subtransactions](https://wiki.postgresql.org/wiki/Autonomous_subtransactions)





## 第 8 章

# PostgreSQL 并发控制系统的实现——封锁

PostgreSQL 使用锁控制并发，锁分为两种类型，一种是系统锁，一种是事务锁。数据库的并发控制技术主要基于封锁并发控制技术，用两阶段锁或严格两阶段锁实现并发控制，这样的锁属于事务锁。很多数据库系统如 Oracle、Informix、MySQL 的 InnoDB 都采用了两阶段或严格两阶段封锁技术，其中一些系统尽管采用了 MVCC 技术，但是它们主要的并发控制技术还是以两阶段或严格两阶段封锁为主。但是 PostgreSQL 却不是这样，虽然 PostgreSQL 也采用了事务锁，但却是以 MVCC 技术为并发控制技术的主体，辅以部分事务锁，所以对于 PostgreSQL 的并发技术而言，第 9 章是其重点。

本章主要讲述 PostgreSQL 的系统锁和事务锁中，偏于面向用户层的事务锁和 DDL 类型的事务锁。另外还会提及 MVCC 技术中辅助实现元组级的并发访问控制的行级锁，但这样的行级锁只在元组头上施加，没有对应的内存锁表，只用于基于快照的元组可见性判断上。

对于 PostgreSQL 而言，简略地讲：PostgreSQL 利用 SS2PL 技术实现了 DDL 的并发控制，利用 MVCC 技术实现了 DQL 和 DML 的并发控制。

## 8.1 锁的概述

PostgreSQL 利用 SS2PL 技术实现了 DDL 的并发控制，这表明，除了事务管理模型要采用 2PL 外，还需要数据库系统提供严格的强封锁协议。

为了讲述清楚 PostgreSQL 的封锁技术，本节先概要地讲述锁相关的内容和范围。

### 8.1.1 锁操作的本质

锁操作，所起的作用就是防止被锁的对象被并发操作同时修改。从技术本质上看，加锁操作就是为特定对象设置一个标志位，然后通过使用锁的机制（对象上存在标志位则不能



改写，放弃加锁请求或等待锁释放后再进行操作）和释放锁（取消特定对象上被设置的标志位），一共三个过程，共同完成排斥其他并发操作对这个特定对象进行操作。

例如，Linux 系统上使用 pthread\_mutex\_t 定义一个名为 “mutex\_lock” 的对象，加锁一次，此对象的内容如下，其中 “\_\_lock = 1” 表示进行了一次 pthread\_mutex\_lock() 操作：

```
mutex_lock = {
    __data = {__lock = 1, __count = 0, __owner = 0, __nusers = 0, __kind = 0, __spins = 0,
    __list = {__prev = 0x0, __next = 0x0}
    },
    __size = "\001\000\000\000\000\000\000\000\000f\373\000\000\001", '\000' <repeats 26 times>,
    __align = 0}
```

8.1.2 与锁相关的文件

PostgreSQL 按锁的体量定义了五种锁。其中，从轻量到重量，包含三种，分别是：SpinLock、LWLock、RegularLock。这些锁，是锁的物理实现方式。前两种是系统级的锁，后一种是事务级的锁，另外，PostgreSQL 定义有另外两种锁，一种是 advisory lock，用以实现用户级的封锁控制，粒度可粗可细，完全由用户控制；第二种是 SIREAD lock，名为谓词锁，是专门用于实现隔离级别中的可串行化技术的。

这些锁，定义和实现在如表 8-1 所示的文件中。这些锁的一些差别，如表 8-2 所示。

表 8-1 PostgreSQL 锁操作相关的文件

文件名	功能
s_lock.h/s_lock.c	Hardware-dependent implementation of spinlocks. 物理 CPU 支持 test-and-set (TAS) 指令，则依赖物理机器的 TAS 指令
spin.h/spin.c	Hardware-independent implementation of spinlocks. 物理 CPU 不支持 test-and-set (TAS) 指令，则依赖 PGSemaphores 实现 spinlock，软件实现的方式，速度会慢许多
latch.h/latch.c	Routines for inter-process latches, 系统锁的实现
lwlock.h/lwlock.c	Lightweight lock manager. 轻量级锁的功能实现，包括加锁和释放锁等操作
lockdefs.h	Frontend exposed parts of postgres' low level lock mechanism PostgreSQL 提供的 RegularLock
lockfuncs.c	Functions for SQL access to various lock-manager capabilities.
lock.h/lock.c	POSTGRES primary lock mechanism
lockcmds.h/lockcmds.c	prototypes for lockcmds.c/LOCK command support code 对 “LOCK TABLE...” 命令提供支持
lockoptions.h	Common header for some locking-related declarations. 定义有 LockWaitPolicy 等部分内容
lmgr.h/lmgr.c	POSTGRES lock manager code. 锁管理器
deadlock.c	POSTGRES deadlock detection code. 死锁检测
predicate.c/ predicate.h	predicate locking to support full serializable transaction isolation 谓词锁，实现事务的可串行化操作



表 8-2 各种锁的差异对比表

锁类型	速度	等待队列	死锁检测	加锁对象
SpinLock	最快	无	不需要	数据库的系统级操作：如做进程的切换操作、共享内存的操作、内存使用的 hash 表操作、缓冲区的获取和释放、CheckPoint 的操作、日志相关操作等
LWLock	快	有	没有	数据库的系统级操作
ReguarLock	—	有	有	事务级别的锁，对数据库对象如表、视图等加锁；也包括页面级和元组级的锁，都归于这两个类型中。但元组级的锁和 MVCC 技术紧密关联
advisory lock	—	有	没有	用户操作的对象，由用户控制
SIReadLocks	—	有	有	在可串行化隔离级别下，被读过的数据项所在的事务，持有谓词读锁的信息，详情参见第 9 章

### 8.1.3 与锁相关的内存初始化

PostgreSQL 通过 CreateSharedMemoryAndSemaphores() 函数创建共享内存和信号量。共享内存是一个多种类型内存的集合，包括缓冲区内内存、日志使用的内存、事务操作使用的内存、各种锁相关的内存等。其中，与锁相关的内容如下：

```
void
CreateSharedMemoryAndSemaphores(bool makePrivate, int port) //创建共享内存
{
    ...
    if (!IsUnderPostmaster) //系统处于启动状态
    {
        ...
        size = 100000;
        size = add_size(size, SpinlockSemaSize());
        //SpinLock信号量将被初始化，用于只能使用semaphore模拟SpinLock的情况
        ...
        size = add_size(size, LockShmemSize());
        //计算用于锁共享缓存的锁表的大小，包括：lock hash table、proclock hash table
        size = add_size(size, PredicateLockShmemSize());
        //计算用于谓词锁表的大小，包括：predicate lock hash table、rw-conflict pool等
        ...
        size = add_size(size, LWLockShmemSize()); //计算LWLock锁表的大小
        ...
    }
    else
    {
        ...
    }
    ...
    // Now initialize LWLocks, which do shared memory allocation and are needed
    for InitShmemIndex.
    CreateLWLocks(); //创建LWLock，即初始化LWLock，包括“MainLWLockArray”
    ...
    InitLocks(); //初始化锁管理器，包括：LockMethodLockHash（位于共享内存）、
```



```

    LockMethodProcLockHash ( 位于共享内存 )、LockMethodLocalHash ( 位于进程的局部内存 )
    InitPredicateLocks ();      //初始化用于实现SSI技术的谓词锁表
    ...
    CheckpointerShmemInit (); //CheckPoint初始化
    ...
}

```

## 8.2 系统锁

PostgreSQL 提供了多种系统级别的锁，包括 SpinLock、LwLock。对于 SpinLock，PostgreSQL 没有使用操作系统的 Mutex 来保证共享数据的完整性，而是使用自定义的 SpinLock 和基于 SpinLock 的 Latch 互斥并发。

下面我们将对 SpinLock 和 LwLock 进行详细讨论。

### 8.2.1 SpinLock

SpinLock（自旋锁）的特点是依赖物理硬件，没有等待队列和死锁检测，这样使得加锁速度会很快。所以适合使用在封锁时间很短的情况下，如系统自身共享资源的并发保护需要使用 SpinLock。但是如果存在加锁 SpinLock 冲突，则后申请加锁者会忙等直到获得锁，或者锁超时。

本节从 SpinLock 加锁的本质、SpinLock 实现的技术、SpinLock 加锁与释放锁、SpinLock 在 PostgreSQL 中的作用四个角度，来探讨 SpinLock 所涉及的内容。

PostgreSQL 建议：如果一个锁被持有超过几十个指令，不要使用自旋锁。

#### 1. SpinLock 加锁的本质

SpinLock 的锁操作，符合锁操作的技术本质，加锁是设置一个特定的标志位，释放锁是取消标识位。如下以 CheckPoint 相关操作为例，进行说明，其他类似。

PostgreSQL 定义 CheckPoint 结构体 CheckpointerShmemStruct 如下：

```

typedef struct
{
    pid_t      checkpointer_pid; /* PID (0 if not started) */
    slock_t    ckpt_lck;
    /* protects all the ckpt_* fields */ //CheckpointerShmemStruct加锁的标识位
    ...
    CheckpointerRequest requests[FLEXIBLE_ARRAY_MEMBER];
} CheckpointerShmemStruct; //CheckPoint结构体的名称

```

而 slock\_t 的定义如下（不同硬件，不完全相同，只举 2 个例子）：

```

typedef int slock_t; //硬件平台为ARM、ARM64

```

或者:

```
typedef unsigned char slock_t; //硬件平台为AMD Opteron、Intel EM64T
```

无论是 ARM 还是 AMD 等硬件设备, `slock_t` 的定义是不同的, 但是对于上层而言, 它就是一个标志位, 用于被设置或取消设置, 表示“有”(被设置)和“无”(取消设置)的概念。所以, 从 `CheckpointShmemStruct` 结构体看, 其成员“`ckpt_lck`”作为一个标识, 用以保护全局唯一的 `CheckpointShmemStruct` 结构体避免被多个进程并发修改。

```
static CheckpointerShmemStruct *CheckpointShmem; //被定义为static, 表示全局唯一
```

而 `CheckpointShmemStruct` 结构体的初始化, 是通过 `CheckpointShmemInit` 函数在共享内存中完成, 换句话说, 上面我们在 `CheckpointShmemStruct` 结构体中提到的“`ckpt_lck`”, 就是位于内存中的一块区域(或一个 `char` 大小或一个 `int` 大小等), 这块区域用于标识是否用以保护 CheckPoint 相关操作。也就是说, 锁是内存中的一块区域, 被用于置位(工程实现中, 内存被初始化后, 通常全部是“0”, 所以置位指的是把 0 变为 1, 即完成加锁)或取消置位(释放锁, 重新置为“0”)。

```
CheckpointShmemInit(void) //在共享内存中创建CheckPoint结构体
{
    Size    size = CheckpointerShmemSize();
    ...
    CheckpointerShmem = (CheckpointerShmemStruct *) ShmemInitStruct("Checkpoint
    Data", size, &found);
    ...
}
```

## 2. SpinLock 实现的技术

实现 SpinLock 自旋锁的方式, 依赖于物理 CPU 是否支持 test-and-set (TAS) 指令(如果不支持则采用信号量 semaphore 模拟的方式, 速度会很慢, 可以参考 `pg_sema.h`)。

例如, 物理硬件是 AMD Opteron、Intel EM64T 等 CPU (其他类型的 CPU 对应不同的代码, 相关代码参见 `s_lock.h` 文件), PostgreSQL 首先定义如下汇编代码, 以实现最底层的加锁操作, 之后, 上层代码就可以直接调用基于 `tas(volatile slock_t *lock)` 函数的宏定义 `TAS(lock)`, 实现基本的 spinlock。

```
typedef unsigned char slock_t; //定义一个char类型, 用于与CPU进行数据交换
#define TAS(lock) tas(lock)
//上层统一使用TAS(lock)宏定义屏蔽不同CPU下定义的tas(volatile slock_t *lock)

static __inline__ int //TAS实现方式之一
tas(volatile slock_t *lock)
//测试是否可以在输入参数“lock”加锁, 如果加锁成功, 返回0并将“lock”置为1
{
```



```

register slock_t _res = 1;
/* __asm__ "表示后面的代码为内嵌汇编, "__volatile__"表示编译器不要优化代码, 后面的指令保留原样 */
__asm__ __volatile__(
    "    lock                \n"
    "    xchgb    %0,%1    \n"
:
    "+q"(_res), "+m"(*lock)
:
    /* no inputs */
:
    "memory", "cc"); // "memory"表明: 不要将该段内嵌汇编指令与前面的指令重新排序,
                    // 不要将变量缓存到寄存器
    return (int) _res;
}
...
#ifdef HAVE_GCC__SYNC_INT32_TAS
...
static __inline__ int
tas(volatile slock_t *lock) // TAS实现方式之二, 还有多种实现方式, 多种TAS的实现方式参见s_lock.h文件
{
    return __sync_lock_test_and_set(lock, 1);
}

```

再往上一层, 宏 `S_LOCK (lock)` 基于宏定义 `TAS(lock)` 做加锁操作, 如果加锁失败 (`TAS(lock)` 返回值为 1), 则使用 `s_lock()` 函数再次进行尝试:

```

#if !defined(S_LOCK)
#define S_LOCK(lock) \
    (TAS(lock) ? s_lock((lock), __FILE__, __LINE__, PG_FUNCNAME_MACRO) : 0)
#endif /* S_LOCK */

```

`s_lock()` 函数, 将有时延, 花费更多的时间, 导致加锁的速度会变慢:

```

int
s_lock(volatile slock_t *lock, const char *file, int line, const char *func)
{
    SpinDelayStatus delayStatus;
    init_spin_delay(&delayStatus, file, line, func);

    while (TAS_SPIN(lock))
    {
        perform_spin_delay(&delayStatus); // 延迟一段时间, 再次尝试加锁
    }
    finish_spin_delay(&delayStatus);
    return delayStatus.delays;
}

```

函数 `s_lock()` 使用了 `TAS_SPIN`, 从 PostgreSQL 官方的解释可以区分 `TAS_SPIN()` 与 `TAS()` 二者的区别:

```

* Usually, S_LOCK() is implemented in terms of even lower-level macros
// 请注意阅读所引用的注释, 列在本书的注释, 都是重要的值得精读的

```



```

*   TAS() and TAS_SPIN():
*
*   int TAS(slock_t *lock)
*       Atomic test-and-set instruction. Attempt to acquire the lock,
*       but do *not* wait. Returns 0 if successful, nonzero if unable
*       to acquire the lock.
*
*   int TAS_SPIN(slock_t *lock)
*       Like TAS(), but this version is used when waiting for a lock
*       previously found to be contended. By default, this is the
*       same as TAS(), but on some architectures it's better to poll a
*       contended lock using an unlocked instruction and retry the
*       atomic test-and-set only when it appears free.
*
*   TAS() and TAS_SPIN() are NOT part of the API, and should never be called
*   directly.

```

### 3. SpinLock 加锁与释放锁

对于 PostgreSQL 而言, SpinLock 的使用分为四种方式, 分别为: 初始化、加锁、解锁、释放锁空间, 这些操作, 是通过 SpinLockAcquire 和 SpinLockRelease 等宏定义调用相应的函数完成的。

```

#define SpinLockInit(lock)    S_INIT_LOCK(lock) //初始化
#define SpinLockAcquire(lock) S_LOCK(lock)      //加锁
#define SpinLockRelease(lock) S_UNLOCK(lock)    //解锁, 即释放锁
#define SpinLockFree(lock)   S_LOCK_FREE(lock)  //释放锁的内存空间

```

SpinLock 实现技术主要分析了 SpinLock 的加锁操作, 这里重点分析锁的初始化、锁空间释放和解锁操作, 先看解锁操作。

解锁操作与加锁操作动作正好相反, 是要把内存置为“0”, 例如, 在 PowerPC 机器上, 解锁代码如下:

```

#ifdef USE_PPC_LWSYNC
#define S_UNLOCK(lock)    \
do \
{ \
    __asm__ __volatile__ ("    lwsync \n" ::: "memory"); \
    *((volatile slock_t *) (lock)) = 0; \
} while (0)
#else
#define S_UNLOCK(lock)    \
do \
{ \
    __asm__ __volatile__ ("    sync \n" ::: "memory"); \
    *((volatile slock_t *) (lock)) = 0; \
} while (0)

```

```

} while (0)
#endif /* USE_PPC_LWSYNC */

```

SpinLock 锁的初始化，同加锁和释放锁一样，都依赖于不同的硬件设备，比如，不支持 TAS 指令的机器只能用信号量来模拟 SpinLock 时，其锁初始化的方式如下：

```

void
s_init_lock_sema(volatile slock_t *lock, bool nested)
{
    static int    counter = 0;
    *lock = ((++counter) % NUM_SPINLOCK_SEMAPHORES) + 1; //用一个不断增长的计数器置值
}

```

SpinLock 锁空间释放，没有被任何代码调用，这是因为系统级使用的 SpinLock 在系统中已经预先定义好，在系统运行期间一直存在，不需要释放。

#### 4. SpinLock 在 PostgreSQL 中的作用

SpinLock 加锁的操作比较短暂，比如，申请执行一个 CheckPoint 的操作中多次用到 SpinLock，快速加锁，加锁之后只执行极其简洁的少数个操作，然后快速释放锁：

```

void
RequestCheckpoint(int flags)
//申请执行一个CheckPoint，存在为“CheckpointShmem”的元素多次操作进行加锁的可能
{...
    SpinLockAcquire(&CheckpointShmem->ckpt_lck); //申请加锁
    old_failed = CheckpointShmem->ckpt_failed; //快速获取已经存在的旧值
    old_started = CheckpointShmem->ckpt_started;
    CheckpointShmem->ckpt_flags |= flags;
    SpinLockRelease(&CheckpointShmem->ckpt_lck); //快速释放，加锁阶段执行的操作很好
    ...
    if (flags & CHECKPOINT_WAIT) //如果需要等待CheckPoint完成
    {...
        for (;;)
        {
            SpinLockAcquire(&CheckpointShmem->ckpt_lck); //申请加锁
            new_started = CheckpointShmem->ckpt_started; //快速获取已经存在的旧值
            SpinLockRelease(&CheckpointShmem->ckpt_lck);
            //快速释放，加锁阶段执行的操作很好
            ...
        }
        for (;;) //We are waiting for ckpt_done >= new_started, in a modulo sense.
        {...
            SpinLockAcquire(&CheckpointShmem->ckpt_lck); //申请加锁
            new_done = CheckpointShmem->ckpt_done; //快速获取已经存在的旧值
            new_failed = CheckpointShmem->ckpt_failed;
            SpinLockRelease(&CheckpointShmem->ckpt_lck);

```



```
//快速释放, 加锁阶段执行的操作很好
```

```
...
    }
...
}
}
```

PostgreSQL 主要封锁数据库的系统级操作, 如做进程的切换操作、共享内存的操作、内存使用的 hash 表操作、缓冲区的获取和释放、CheckPoint 的操作、日志相关操作等。详细情况如表 8-3 所示。

表 8-3 SpinLock 封锁的对象

编号	对象与 SpinLock 锁	说明
1	CheckpointShmemStruct[ckpt_lck]	保护 Checkpointer
2	FreeListData[mutex]	用以保护快速访问内存对象的动态 HASH 表
3	BufferStrategyControl[buffer_strategy_lock]	保护缓存策略表
4	FastPathStrongRelationLockData[mutex]	a fast-path interface to send simple function calls to the server.
5	ShmemLock 自身就是一个 SpinLock	We use the ShmemLock spinlock to protect LWLockCounter
6	FixedParallelState[mutex]	Fixed-size parallel state
7	pgssEntry[mutex]	Statistics per statement
8	pgssSharedState[mutex]	Global shared state
9	ProcStructLock 自身就是一个 SpinLock	This spinlock protects the freelist of recycled PGPROC structures. We cannot use an LWLock because the LWLock manager depends on already having a PGPROC and a wait semaphore
10	ProcArrayStruct[known_assigned_xids_lck]	protects head/tail pointers
11	ParallelHeapScanDescData[phs_mutex]	Shared state for parallel heap scan
12	shm_mq[mq_mutex]	shared memory message queue
13	shm_toc[toc_mutex]	shared memory segment table of contents
14	SISeg[msgnumLock]	Shared cache invalidation memory segment
15	ReplicationSlot[mutex]	Shared memory state of a single replication slot
16	OldSnapshotControlData[mutex_current/ mutex_latest_xmin/mutex_threshold]	Structure for dealing with old_snapshot_threshold implementation
17	test_shm_mq_header[mutex]	determine whether all workers started up OK and successfully attached to their respective shared message queues
18	WalRcvData[mutex]	Shared memory area for management of walreceiver process
19	WalSnd[mutex]	Each walsender has a WalSnd struct in shared memory
20	XLogCtlInsert[insertpos_lck]	Shared state data for WAL insertion
21	XLogCtlData[ulsn_lck/info_lck]	Total shared-memory state for XLOG
22	test_lock_struct[lock]	test program for verifying a port's spinlock support
23	dummy_spinlock 自身就是一个 SpinLock	Initialize dummy_spinlock, in case we are on a platform where we have to use the fallback implementation of pg_memory_barrier()



## 8.2.2 LWLock

LWLock（轻量锁）主要用于封锁共享内存中的数据结构，以达到互斥访问的目的（避免并发造成的同一份数据被不同进程改写带来的不一致，即一段关键代码同时只允许一个进程进行改写）。

LWLock 依赖 SpinLock 实现，有等待队列但没有死锁检测，能自动释放锁（采用 elog 报错机制，锁管理器可以释放 LWLock 锁）。但是没有超时机制。

LWLock 有读和写两种模式，即共享和排它两种模式。

LWLock 锁的施加和释放时间很快，所以适合使用在封锁时间较短的情况下。

本节从 LWLock 加锁的本质、LWLock 实现的技术、LWLock 加锁与释放锁、LWLock 在 PostgreSQL 中的作用等多个角度，来探讨 LWLock 所涉及的内容。

### 1. 预定义的 LWLock 锁

预定义的 LWLock 锁有 42 个，具体如下：

```
#define ShmemIndexLock (&MainLWLockArray[1].lock) //共享内存锁
#define OidGenLock (&MainLWLockArray[2].lock) //生成Oid的值的锁
#define XidGenLock (&MainLWLockArray[3].lock) //生成Xid（事务号）的值的锁
#define ProcArrayLock (&MainLWLockArray[4].lock)
//进程锁（PostgreSQL是多进程体系结构，用于进程间互斥）
#define SInvalReadLock (&MainLWLockArray[5].lock)
//共享cache的读锁(SI: shared cache invalidation data manager)
#define SInvalWriteLock (&MainLWLockArray[6].lock) //共享cache的写锁
#define WALBufMappingLock (&MainLWLockArray[7].lock)
//预先日志（WAL）缓存区锁，把数据写入缓存区
#define WALWriteLock (&MainLWLockArray[8].lock)
//预先日志（WAL）写锁，把数据从缓存区刷出到持久化存储
#define ControlFileLock (&MainLWLockArray[9].lock) //控制文件锁
#define CheckpointLock (&MainLWLockArray[10].lock) //CheckPoint锁，用于创建CheckPoint
#define CLogControlLock (&MainLWLockArray[11].lock) //事务日志（CLog）锁
#define SubtransControlLock (&MainLWLockArray[12].lock) //子事务锁
#define MultiXactGenLock (&MainLWLockArray[13].lock) //多事务生成锁
#define MultiXactOffsetControlLock (&MainLWLockArray[14].lock) //多事务偏移控制锁
#define MultiXactMemberControlLock (&MainLWLockArray[15].lock) //多事务成员控制锁
#define RelCacheInitLock (&MainLWLockArray[16].lock) //RelationCache初始化锁
#define CheckpointerCommLock (&MainLWLockArray[17].lock)
//CheckPoint常规锁，用于创建CheckPoint之外的CheckPoint操作
#define TwoPhaseStateLock (&MainLWLockArray[18].lock) //2阶段提交状态锁，用于分布式事务
#define TablespaceCreateLock (&MainLWLockArray[19].lock)
//创建、删除表空间时以排他模式加的锁
#define BtreeVacuumLock (&MainLWLockArray[20].lock)
//在BTree上执行Vacuum操作清理无用元组时加的锁
#define AddinShmemInitLock (&MainLWLockArray[21].lock) //共享内存初始化锁
```

```

#define AutovacuumLock (&MainLWLockArray[22].lock) //自动执行Vacuum的进程执行操作时的锁
#define AutovacuumScheduleLock (&MainLWLockArray[23].lock)
//自动执行Vacuum的进程执行do_autovacuu()函数时的锁
#define SyncScanLock (&MainLWLockArray[24].lock)
//在heap上进行扫描定位数据位置(BlockNumber)时的互斥锁
#define RelationMappingLock (&MainLWLockArray[25].lock)
//RelationMapping锁(Catalog-to-filenode mapping)
#define AsyncCtlLock (&MainLWLockArray[26].lock)
//异步控制锁(Asynchronous notification: NOTIFY, LISTEN, UNLISTEN)
#define AsyncQueueLock (&MainLWLockArray[27].lock) //异步队列锁
#define SerializableXactHashLock (&MainLWLockArray[28].lock)
//SSI(Serializable Snapshot Isolation)相关的锁
#define SerializableFinishedListLock (&MainLWLockArray[29].lock) //SSI相关的锁
#define SerializablePredicateLockListLock (&MainLWLockArray[30].lock) //SSI相关的锁
#define OldSerXidLock (&MainLWLockArray[31].lock) //SSI相关的锁
#define SyncRepLock (&MainLWLockArray[32].lock) //复制功能,同步复制相关锁
#define BackgroundWorkerLock (&MainLWLockArray[33].lock)
//PostgreSQL 9.4支持background workers后台进程动态注册、启动、停止,是并行化的基础
#define DynamicSharedMemoryControlLock (&MainLWLockArray[34].lock)
//动态共享内存控制锁
#define AutoFileLock (&MainLWLockArray[35].lock)
//修改"postgresql.auto.conf"文件的操作锁
#define ReplicationSlotAllocationLock (&MainLWLockArray[36].lock)
//复制功能,一个slot分配互斥锁
#define ReplicationSlotControlLock (&MainLWLockArray[37].lock)
//复制功能,一个slot查找互斥锁
#define CommitTsControlLock (&MainLWLockArray[38].lock) //事务提交时间控制锁,与页面有关
#define CommitTsLock (&MainLWLockArray[39].lock)
//事务提交时间锁,与上一个锁分别锁事务提交过程中不同的对象
#define ReplicationOriginLock (&MainLWLockArray[40].lock) //复制功能,复制相关锁
#define MultiXactTruncationLock (&MainLWLockArray[41].lock) //多事务Truncate锁
#define OldSnapshotTimeMapLock (&MainLWLockArray[42].lock) //老快照时间映射锁

```

PostgreSQL 的文档中,在“wait\_event Description”一节提供了一个表格,对各种锁作了描述,如表 8-4 所示,可以和预定义的 42 个 LWLock 锁对应着互相理解。

表 8-4 PostgreSQL 文档中对 LWLock (轻量锁) 的描述

Wait Event Type	Wait Event Name	Description
LWLockNamed <sup>①</sup>	ShmemIndexLock	Waiting to find or allocate space in shared memory.
	OidGenLock	Waiting to allocate or assign an OID.
	XidGenLock	Waiting to allocate or assign a transaction id.
	ProcArrayLock	Waiting to get a snapshot or clearing a transaction id at transaction end.
	SInvalReadLock	Waiting to retrieve or remove messages from shared invalidation queue.



(续)

Wait Event Type	Wait Event Name	Description
LWLockNamed <sup>①</sup>	SInvalWriteLock	Waiting to add a message in shared invalidation queue.
	WALBufMappingLock	Waiting to replace a page in WAL buffers.
	WALWriteLock	Waiting for WAL buffers to be written to disk.
	ControlFileLock	Waiting to read or update the control file or creation of a new WAL file.
	CheckpointLock	Waiting to perform checkpoint.
	CLogControlLock	Waiting to read or update transaction status.
	SubtransControlLock	Waiting to read or update sub-transaction information.
	MultiXactGenLock	Waiting to read or update shared multixact state.
	MultiXactOffsetControlLock	Waiting to read or update multixact offset mappings.
	MultiXactMemberControlLock	Waiting to read or update multixact member mappings.
	RelCacheInitLock	Waiting to read or write relation cache initialization file.
	CheckpointInterCommLock	Waiting to manage fsync requests.
	TwoPhaseStateLock	Waiting to read or update the state of prepared transactions.
	TablespaceCreateLock	Waiting to create or drop the tablespace.
	BtreeVacuumLock	Waiting to read or update vacuum-related information for a Btree index.
	AddinShmemInitLock	Waiting to manage space allocation in shared memory.
	AutovacuumLock	Autovacuum worker or launcher waiting to update or read the current state of autovacuum workers.
	AutovacuumScheduleLock	Waiting to ensure that the table it has selected for a vacuum still needs vacuuming.
	SyncScanLock	Waiting to get the start location of a scan on a table for synchronized scans.
	RelationMappingLock	Waiting to update the relation map file used to store catalog to filenode mapping.
	AsyncCtlLock	Waiting to read or update shared notification state.
	AsyncQueueLock	Waiting to read or update notification messages.
	SerializableXactHashLock	Waiting to retrieve or store information about serializable transactions.
	SerializableFinishedListLock	Waiting to access the list of finished serializable transactions.



(续)

Wait Event Type	Wait Event Name	Description
LWLockNamed <sup>①</sup>	SerializablePredicateLockListLock	Waiting to perform an operation on a list of locks held by serializable transactions.
	OldSerXidLock	Waiting to read or record conflicting serializable transactions.
	SyncRepLock	Waiting to read or update information about synchronous replicas.
	BackgroundWorkerLock	Waiting to read or update background worker state.
	DynamicSharedMemoryControlLock	Waiting to read or update dynamic shared memory state.
	AutoFileLock	Waiting to update the postgresql.auto.conf file.
	ReplicationSlotAllocationLock	Waiting to allocate or free a replication slot.
	ReplicationSlotControlLock	Waiting to read or update replication slot state.
	CommitTsControlLock	Waiting to read or update transaction commit timestamps.
	CommitTsLock	Waiting to read or update the last value set for the transaction timestamp.
	ReplicationOriginLock	Waiting to setup, drop or use replication origin.
	MultiXactTruncationLock	Waiting to read or truncate multixact information.
LWLockTranche <sup>②</sup>	clog	Waiting for I/O on a clog (transaction status) buffer.
	commit_timestamp	Waiting for I/O on commit timestamp buffer.
	subtrans	Waiting for I/O a subtransaction buffer.
	multixact_offset	Waiting for I/O on a multixact offset buffer.
	multixact_member	Waiting for I/O on a multixact_member buffer.
	async	Waiting for I/O on an async (notify) buffer.
	oldserxid	Waiting to I/O on an oldserxid buffer.
	wal_insert	Waiting to insert WAL into a memory buffer.
	buffer_content	Waiting to read or write a data page in memory.
	buffer_io	Waiting for I/O on a data page.
	replication_origin	Waiting to read or update the replication progress.
	replication_slot_io	Waiting for I/O on a replication slot.
	proc	Waiting to read or update the fast-path lock information.
	buffer_mapping	Waiting to associate a data block with a buffer in the buffer pool.
	lock_manager	Waiting to add or examine locks for backends, or waiting to join or exit a locking group (used by parallel query).
	predicate_lock_manager	Waiting to add or examine predicate lock information.

(续)

Wait Event Type	Wait Event Name	Description
Lock <sup>③</sup>	relation	Waiting to acquire a lock on a relation.
	extend	Waiting to extend a relation.
	page	Waiting to acquire a lock on page of a relation.
	tuple	Waiting to acquire a lock on a tuple.
	transactionid	Waiting for a transaction to finish.
	virtualxid	Waiting to acquire a virtual xid lock.
	speculative token	Waiting to acquire a speculative insertion lock.
	object	Waiting to acquire a lock on a non-relation database object.
	userlock	Waiting to acquire a userlock.
	advisory	Waiting to acquire an advisory user lock.
BufferPin <sup>④</sup>	BufferPin	Waiting to acquire a pin on a buffer.

- ① The backend is waiting for a specific named lightweight lock. Each such lock protects a particular data structure in shared memory. wait\_event will contain the name of the lightweight lock.
- ② The backend is waiting for one of a group of related lightweight locks. All locks in the group perform a similar function; wait\_event will identify the general purpose of locks in that group.
- ③ The backend is waiting for a heavyweight lock. Heavyweight locks, also known as lock manager locks or simply locks, primarily protect SQL-visible objects such as tables. However, they are also used to ensure mutual exclusion for certain internal operations such as relation extension. wait\_event will identify the type of lock awaited.
- ④ The server process is waiting to access to a data buffer during a period when no other process can be examining that buffer. Buffer pin waits can be protracted if another process holds an open cursor which last read data from the buffer in question

2. LWLock 封锁的本质

LWLock 的锁操作，符合锁操作的技术本质，加锁同样是设置一个特定的标志位，释放锁是取消标识位。如下以 CheckPoint 相关操作为例，进行说明，其他类似。

PostgreSQL 定义 LWLock 结构体如下：

```
typedef struct LWLock //LWLock锁的数据结构，绑定了锁、锁的属主、锁的等待者这三者，
PostgreSQL 9.4.x之前的版本的实现方式有较大不同
{
    uint16         tranche;           /* tranche ID */
    //LWLock上多了事务号，表明实施加锁操作的属主
    pg_atomic_uint32 state;           /* state of exclusive/nonexclusive lockers */
    //锁对象，一个标志对象，有多个标志位（通常是无符号32位数），即有32个标志位，不同标志位表达
    //不同含义。之所以这样定义，是因为这样的锁需要同时表达多种含义（如设置为LW_FLAG_HAS_WAITERS
    //同时还可设置为）
    dlist_head     waiters;           /* list of waiting PGPROCS */
    //本锁的等待队列，表明那些进程在等待本锁释放
#ifdef LOCK_DEBUG
```

```

    pg_atomic_uint32 nwaiters;    /* number of waiters */ //等待者的数量
    struct PGPROC *owner;        /* last exclusive owner of the lock */ //锁的属主
#endif
} LWLock;

```

PostgreSQL 定义了不同的标识符号, 用以为“pg\_atomic\_uint32 state”标识不同的含义, 已经在 PostgreSQL 中定义好的标识符号如下:

```

#define LW_FLAG_HAS_WAITERS    ((uint32) 1 << 30) //标志, LWLock锁有等待者
#define LW_FLAG_RELEASE_OK    ((uint32) 1 << 29)
//标志, LWLock锁空闲, 即没有施加锁 (state值为“536870912”)
#define LW_FLAG_LOCKED        ((uint32) 1 << 28)
//标志, LWLock锁存在, 即已被加锁 (state值为“268435456”)

#define LW_VAL_EXCLUSIVE        ((uint32) 1 << 24)
//锁的模式, LWLock排它锁模式, 加锁的标志位之一, 供加锁操作时使用
#define LW_VAL_SHARED          1
//锁的模式, LWLock共享锁模式, 加锁的标志位之一, 供加锁操作时使用
#define LW_LOCK_MASK            ((uint32) ((1 << 25)-1)) //标志, LWLock锁标志
/* Must be greater than MAX_BACKENDS - which is 2^23-1, so we're fine. */
#define LW_SHARED_MASK          ((uint32) ((1 << 24)-1)) //标志, LWLock存在共享锁

```

PostgreSQL 定义 LWLock 结构体之后, 一个 LWLock 对象就可以出现在其他需要被保护的对象中, 如进程对象 (PGPROC)。

```

struct PGPROC //进程的结构体, PostgreSQL是多进程结构, 此结构体上定义了快速的事务锁,
用以加快事务的操作
{...
    /* Per-backend LWLock.  Protects fields below (but not group fields). */
    LWLock backendLock; //定义LWLock锁, 用于保护进程结构体中的共享资源不受并发操作破坏
    //如下对象被backendLock保护
    /* Lock manager data, recording fast-path locks taken by this backend. */
    uint64 fpLockBits; /* lock modes held for each fast-path slot */
    //快速访问事务锁的技术, 在进程结构体上直接记录事务锁的设置情况, 参见8.4.1节
    Oid fpRelId[FP_LOCK_SLOTS_PER_BACKEND]; /* slots for rel oids */ //同上
    bool fpVXIDLock; /* are we holding a fast-path VXID lock? */
    LocalTransactionId fpLocalTransactionId; /* lxid for fast-path VXID lock */
...};

```

进程对象 (PGPROC) 通过 InitProcGlobal() 函数完成初始化, 然后由此函数调用 LWLockInitialize() 函数完成对进程锁“backendLock”的初始化工作, 即设置锁标志为“LW\_FLAG\_RELEASE\_OK”。

```

void
InitProcGlobal(void)
{...
    for (i = 0; i < TotalProcs; i++)

```



```

{
    /* Set up per-PGPROC semaphore, latch, and backendLock. Prepared xact
     * dummy PGPROCs don't need these though - they're never associated with a real process */
    if (i < MaxBackends + NUM_AUXILIARY_PROCS)
    {
        PGSemaphoreCreate(&(procs[i].sem));
        InitSharedLatch(&(procs[i].procLatch));
        LWLockInitialize(&(procs[i].backendLock), LWTRANCHE_PROC);
        //调用LWLockInitialize()初始化backendLock
    }
    ...
}
...
}

```

而 LWLockInitialize() 函数要给 “state” 成员设置标识位为 “LW\_FLAG\_RELEASE\_OK”。

```

void
LWLockInitialize(LWLock *lock, int tranche_id)
{
    pg_atomic_init_u32(&lock->state, LW_FLAG_RELEASE_OK); //设置无锁标志
#ifdef LOCK_DEBUG
    pg_atomic_init_u32(&lock->nwaiters, 0);
#endif
    lock->tranche = tranche_id;
    dlist_init(&lock->waiters);
}

```

从以上的分析可以看出，PGPROC 结构体的初始化，是通过 LWLockInitialize() 函数在共享内存中对 “backendLock” 等操作完成，而 “backendLock” 的 “state” 是位于内存中的一块区域（无符号 32 个 bit 位的一段内存空间），这块区域用于标识是否用以保护 PGPROC 结构体。也就是说，LWLock 锁是内存中的一块区域，被用于置位，只是与 SpinLock 不同的是，LWLock 锁这块内存区域在不同的 bit 位设置不同的值以表示多种含义。另外，每个 LWLock 中有明确的属主和等待者，这起到了连接已加锁和申请加锁两种角色的作用，便于知晓谁在什么锁上等待谁这样的关系。

### 3. LWLock 实现的技术

上一节中，我们介绍了 LWLock 的数据结构，其中 LWLock 锁的 “state” 上的锁标志可以有两种模式，如下所示：

```

typedef enum LWLockMode
{
    LW_EXCLUSIVE, //排他模式
    LW_SHARED,    //共享模式
    LW_WAIT_UNTIL_FREE /* A special mode used in PGPROC->lwlockMode,

```

```

        * when waiting for lock to become free. Not
        * to be used as LWLockAcquire argument */
} LWLockMode;

```

而 LWLock 结构体定义了 state 标志位 (LWLock 锁的数据结构参见上一节), 此标志位是通过 LWLockAcquire () 对 LWLockAttemptLock() 函数调用完成标志位设置的。

```

LWLockAttemptLock(LWLock *lock, LWLockMode mode)
{...
    old_state = pg_atomic_read_u32(&lock->state); //读出锁的原始值
    while (true)
    {...
        desired_state = old_state; //desired_state上先保持旧值
        if (mode == LW_EXCLUSIVE)
        {
            lock_free = (old_state & LW_LOCK_MASK) == 0;
            if (lock_free)
                desired_state += LW_VAL_EXCLUSIVE;
            //desired_state上先保持旧值后, 根据锁的模式加排它锁的标识
        }
        else
        {
            lock_free = (old_state & LW_VAL_EXCLUSIVE) == 0;
            if (lock_free)
                desired_state += LW_VAL_SHARED;
            //desired_state上先保持旧值后, 根据锁的模式加共享锁的标识
        }
        //desired_state成为在锁标志位上将被设置的值。pg_atomic_compare_exchange_u32()
        完成设置标志位 (不同硬件平台, 设置方式不同)
        if (pg_atomic_compare_exchange_u32(&lock->state, &old_state, desired_state))
        {...}
    }
}
...
}

```

而 pg\_atomic\_compare\_exchange\_u32() 函数调用的其他函数, 因硬件平台而异, 如在 Windows 下, 调用栈如下, 最终使用 InterlockedCompareExchange() 函数完成锁标志位的置位工作, state 标志位被设置为 “LW\_VAL\_SHARED” 或 “LW\_VAL\_EXCLUSIVE”。

```

LWLockAcquire()
    LWLockAttemptLock()
        pg_atomic_compare_exchange_u32()
            pg_atomic_compare_exchange_u32_impl()
                InterlockedCompareExchange() //把目标操作数 (第1参数所指向的内存中的数) 与一个值
                (第3参数) 比较, 如果相等, 则用另一个值 (第2参数) 与目标操作数 (第1参数所指向的内存中的数) 交换

```

另外, 释放锁时也需要将锁标志位置位为 “LW\_FLAG\_RELEASE\_OK”, 且既无



“LW\_VAL\_SHARED”也无“LW\_VAL\_EXCLUSIVE”，置位操作通过 `pg_atomic_sub_fetch_u32()` 函数完成，在 Windows 下的调用栈示例如下所示：

```
LWLockRelease()
    pg_atomic_sub_fetch_u32()
        pg_atomic_sub_fetch_u32_impl()
            pg_atomic_fetch_sub_u32_impl()
                pg_atomic_fetch_add_u32_impl()
                    InterlockedExchangeAdd() //用于对一个32位数值执行加法的原子操作
```

#### 4. LWLock 锁的创建

LWLock 用于系统级共享资源的封锁操作，在系统运行期间，系统级的资源需要加锁，实施操作后，被释放，如缓存区封锁的相关操作即如此。所以 LWLock 锁需要在整个数据库 Server 启动期间完成 LWLock 的准备工作，即被创建好然后再被使用（加锁或解锁）。`CreateLWLocks()` 函数完成共享内存中预留锁的空间的工作，调用 `InitializeLWLocks()` 完成锁的初始化工作。

```
//Allocate shmem space for the main LWLock array and all tranches and initialize
it. We also register all the LWLock tranches here.
void
CreateLWLocks(void) //在共享内存中创建所有的系统级LWLock
{...
    if (!IsUnderPostmaster)
    {
        Size spaceLocks = LWLockShmemSize(); //可以求知LWLock锁需要的空间大小，从中
        可以分析出有多少种系统级LWLock，详情如下
        ...
        ptr = (char *) ShmemAlloc(spaceLocks); //在共享内存中分配空间
        ...
        MainLWLockArray = (LWLockPadded *) ptr; //记录地址
        ...
        /* Initialize all LWLocks */
        InitializeLWLocks(); //在共享内存中分配了空间给系统级的LWLock后，对他们进行初始化
    }
    /* Register all LWLock tranches */
    RegisterLWLockTranches(); //注册即加载已经所有的系统级的LWLock（如"main"、
    "buffer_mapping"、"lock_manager"、"predicate_lock_manager"）
}
```

`CreateLWLocks()` 调用 `LWLockShmemSize()` 求锁使用的空间大小。

```
Size
LWLockShmemSize(void)
{...
    int    numLocks = NUM_FIXED_LWLOCKS; //PostgreSQL提供的固定的LWLock个数
```



```

numLocks += NumLWLocksByNamedTranches(); //获得“a group of related lightweight
locks”的个数（也是预定义好的LWLock）
/* Space for the LWLock array. */
size = mul_size(numLocks, sizeof(LWLockPadded)); //取2个参数的乘积，每个LWlock的
结构体是LWLockPadded定义的，在LWLock上又封装一层

/* Space for dynamic allocation counter, plus room for alignment. */
size = add_size(size, sizeof(int) + LWLOCK_PADDED_SIZE);

/* space for named tranches. */ //获得“a specific named lightweight lock”的
个数（也是预定义好的LWLock）
size = add_size(size, mul_size(NamedLWLockTrancheRequests,
sizeof(NamedLWLockTranche)));

/* space for name of each tranche. */
for (i = 0; i < NamedLWLockTrancheRequests; i++) //NamedLWLockTrancheRequests个
size = add_size(size, strlen(NamedLWLockTrancheRequestArray[i].tranche_name) + 1);
...
return size;
}

```

PostgreSQL 包含有很多的 LWLock，具体参见表 8-5。

表 8-5 PostgreSQL 预定义的 LWLock

锁名称 (个数)	结构体	功能
MainLWLockArray (42)	LWLockPadded	参见“预定义的 LWLock 锁”一节
BUFFER_MAPPING_LWLOCK(128 个)	—	缓存区管理锁 1 个对象上划分出 128 个区域，分别加锁以减少 冲突
LOCK_MANAGER_LWLOCK(16 个)	—	锁管理器锁 (ReguarLock 式锁的锁管理器) 1 个对象上划分出 16 个区域，分别对每个区域 加减锁以减少冲突
PREDICATELOCK_MANAGER_ LWLOCK(16 个)	—	谓词锁，用于 SSI 技术实现事务隔离级别中的 “可串行快照隔离级别” 1 个对象上划分出 16 个区域，分别对每个区域 加减锁以减少冲突
LWLockCounter (LWLOCK_ PADDED_SIZE 大小)	—	Space for dynamic allocation counter, plus room for alignment
NamedTranches	LWLockPadded	space for named tranches
—	—	Space for dynamic allocation counter
NamedLWLockTrancheArray (1 个)	NamedLWLockTranche	命名的“LWLock tranche”
-(NamedLWLockTrancheRequests 个)	—	全部的命名的“LWLock tranche”

## 5. LWLock 加锁

PostgreSQL 的 LWLock 的实现（在早期的版本如 8.2.15）依赖于 SpinLock，即申

请 LWLock 这个轻量级的锁是通过 LWLockAcquire() 直接调用 SpinLockAcquire() 宏实现的。但是，在后期的版本如 PostgreSQL 9.6 中，这个设计发生了一些变化。设计思想依然是采用物理 CPU 的 test-and-set 指令，但是，不再基于 SpinLockAcquire() 宏，而是通过 LWLockAcquire() 函数调用 LWLockAttemptLock() 再调用 pg\_atomic\_compare\_exchange\_u32() 间接调用底层的 s\_lock() 或其他函数（被调用的函数因硬件平台不同而不同），实现 LWLock，函数间的调用关系如图 8-1 所示。

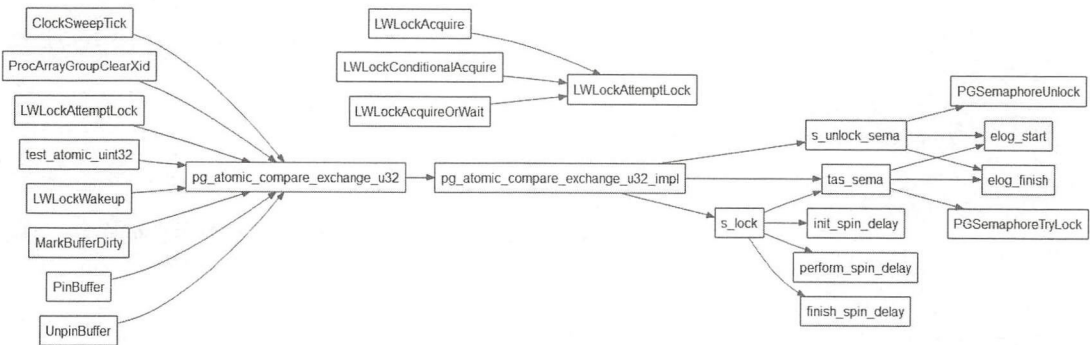


图 8-1 pg\_atomic\_compare\_exchange\_u32() 函数调用关系图

从图 8-1 中可以看出，LWLockAttemptLock() 函数被三个函数调用，这说明 LWLock 锁还存在三种方式，如表 8-6 所示。

表 8-6 LWLock 加锁请求表

函数名	功能
LWLockAcquire()	获取 LWLock 锁，如果不能获得，等待（调用 LWLockReportWaitStart() 函数进入等待队列，直至进程被设置为不再需要等待）直到获取到锁为止
LWLockConditionalAcquire()	获取 LWLock 锁，如果不能获得，不等待
LWLockAcquireOrWait()	获取 LWLock 锁，如果不能获得，等待（调用 LWLockReportWaitStart() 函数进入等待队列，直至进程被设置为不再需要等待）直到想要获取的锁被别的进程释放，但是，不再申请获得该锁。该加锁方式用于改进写“WAL 日志”时多个进程并发写预写日志的效率

下面具体分析一下 LWLock 加锁的功能实现，加锁操作是由 LWLockAcquire() 函数完成，此函数加锁不成功，会多次尝试加锁，如果还是不成功，陷入等待，后被唤醒，再尝试加锁，直至加锁成功。

```
LWLockAcquire(LWLock *lock, LWLockMode mode)
//acquire a lightweight lock(第一个参数lock) in the specified mode(第二个参数)
{...
    AssertArg(mode == LW_SHARED || mode == LW_EXCLUSIVE); //只能通过本函数申请两种加锁模式
    ...
    for (;;) //如不能获得锁，则反复循序。只有第一次或第二次加锁成功才能退出循环
    {...
```



```

mustwait = LWLockAttemptLock(lock, mode);
//试图获取锁, 如果不能获得则不等。这个函数的实现调用了原子操作如pg_atomic_read_u32()
if (!mustwait) //已经获得了锁
{
    LOG_LWDEBUG("LWLockAcquire", lock, "immediately acquired lock");
    break; /* got the lock */ //退出循环
} //否则, 往下继续执行, 意味着没有获得锁
LWLockQueueSelf(lock, mode);
//没有获得锁, 把申请锁的对象(锁的未来属主)加入锁等待队列的队尾
mustwait = LWLockAttemptLock(lock, mode); //尝试第二次获取锁
if (!mustwait) //第二次获取锁的尝试成功, 即加锁成功
{
    LOG_LWDEBUG("LWLockAcquire", lock, "acquired, undoing queue");
    LWLockDequeueSelf(lock);
    //第二次获取锁的尝试成功, 需要把申请锁的对象(已经是锁的属主)从等待队列中去除
    break; //退出循环
}
...

LWLockReportWaitStart(lock);
//第二次获取锁的尝试失败, 则向当前进程汇报加锁失败的消息(给进程结构体上的
wait_event_info赋值)
...

for (;;)
{
    PGSemaphoreLock(&proc->sem);
    //使用信号量阻塞本次加锁请求, 直至被唤醒(可能其他属主释放了本锁)
    if (!proc->lwWaiting) //被唤醒
        break;
    extraWaits++;
}
...

/* Retrying, allow LWLockRelease to release waiters again. */
pg_atomic_fetch_or_u32(&lock->state, LW_FLAG_RELEASE_OK);
//第三次尝试, 看看state上是否已经被释放锁
...

LWLockReportWaitEnd(); //向当前进程汇报加锁操作不再需要等待
// (给进程结构体上的wait_event_info赋值为“0”)
...

/* Now loop back and try to acquire lock again. */
result = false; //循环重新执行, 重新尝试以上的过程
}
...

/* Add lock to list of locks held by this backend */
held_lwlocks[num_held_lwlocks].lock = lock;
//记录加到的锁的信息到当前进程的“锁列表”(held_lwlocks)里
held_lwlocks[num_held_lwlocks++].mode = mode;
...
}

```



下面具体分析一下 LWLock 中无等待队列加锁的功能实现，加锁操作由 LWLockConditionalAcquire() 函数完成，此函数加锁不成功，便不再尝试加锁，直接返回。

```
LWLockConditionalAcquire(LWLock *lock, LWLockMode mode)
{
    ...
    AssertArg(mode == LW_SHARED || mode == LW_EXCLUSIVE);
    //只能通过本函数申请两种加锁模式
    ...
    //相比LWLockAcquire()函数，此处缺少循环，意味着加锁不成功时不会反复尝试
    mustwait = LWLockAttemptLock(lock, mode);
    //试图获取锁，如果不能获得则不等待。这个函数的实现调用了原子操作如pg_atomic_read_u32()
    ...
    return !mustwait; //不管加锁成功还是失败，都不再尝试
}
```

另外，申请加锁的功能有一个特殊的函数实现，加锁操作是由 LWLockAcquireOrWait() 函数完成，此函数的语义是特定的，只用于预先日志封锁的需要，其功能是：如果加锁成功则返回 true；否则，等待此锁被释放但是不再尝试加锁，直接返回 false。

```
/* LWLockAcquireOrWait - Acquire lock, or wait until it's free
 *
 * The semantics of this function are a bit funky. If the lock is currently
 * free, it is acquired in the given mode, and the function returns true. If
 * the lock isn't immediately free, the function waits until it is released
 * and returns false, but does not acquire the lock.
 *
 * This is currently used for WALWriteLock: when a backend flushes the WAL,
 * holding WALWriteLock, it can flush the commit records of many other
 * backends as a side-effect. Those other backends need to wait until the
 * flush finishes, but don't need to acquire the lock anymore. They can just
 * wake up, observe that their records have already been flushed, and return.
 */
bool
LWLockAcquireOrWait(LWLock *lock, LWLockMode mode)
//加锁过程与LWLockAcquire()相似，但语义不同
{
    ...
    //NB: We're using nearly the same twice-in-a-row lock acquisition protocol as
    LWLockAcquire(). Check its comments for details.
    mustwait = LWLockAttemptLock(lock, mode); //试图获取锁，如果不能获得则不等待。这个
    函数的实现调用了原子操作如pg_atomic_read_u32()

    if (mustwait) //加锁不成功，把申请加锁的这个进程加入等待队列
    {
        LWLockQueueSelf(lock, LW_WAIT_UNTIL_FREE); //把申请加锁的这个进程加入等待队列
        mustwait = LWLockAttemptLock(lock, mode); //再次试图获取锁
        if (mustwait)
```

```

{...
    LWLockReportWaitStart(lock);
    //向当前进程汇报加锁失败的消息(给进程结构体上的wait_event_info赋值),表示开始等待
...

    for (;;)
    {
        PGSemaphoreLock(&proc->sem);
        //使用信号量阻塞本次加锁请求,直至被唤醒(可能其他属主释放了本锁)
        if (!proc->lwWaiting)
            break; //被唤醒,则退出循环
        extraWaits++;
    }

    LWLockReportWaitEnd(); //结束等待
...

}
else
{...
    LWLockDequeuesSelf(lock); //第二次加锁成功,则把自己从等待队列中摘除
}
}

...
return !mustwait;
}

```

## 6. LWLock 释放锁

LWLockRelease() 函数释放特定的锁(参数指定的锁), LWLockReleaseAll() 函数调用 LWLockRelease() 函数完成当前进程持有的(通过“held\_lwlocks”持有)所有锁的释放。另外有个 LWLockReleaseClearVar() 函数配对加锁操作中的 LWLockAcquireOrWait() 函数使用。

```

void
LWLockRelease(LWLock *lock) //释放锁并唤醒等待者
{...
    for (i = num_held_lwlocks; --i >= 0;)
    //准备从held_lwlocks中释放锁,为此锁找到当初的加锁模式
    {
        if (lock == held_lwlocks[i].lock)
        {
            mode = held_lwlocks[i].mode;
            break;
        }
    }
...
    if (mode == LW_EXCLUSIVE) //释放锁,即给锁标志位置位(还原为初始值)
        oldstate = pg_atomic_sub_fetch_u32(&lock->state, LW_VAL_EXCLUSIVE);
}

```



```

else
    oldstate = pg_atomic_sub_fetch_u32 (&lock->state, LW_VAL_SHARED);
...
if (check_waiters)
{
    /* XXX: remove before commit? */
    LOG_LWDEBUG("LWLockRelease", lock, "releasing waiters");
    LWLockWakeup(lock); //如果此锁上有等待这，唤醒等待者（告诉操作系统重新调度进程）
}
...
}

```

## 7. LWLock 在 PostgreSQL 中的作用

PostgreSQL 中，LWLock 锁可以加在特定的对象上，如对某个进程加锁，以获取其上锁的状态信息：

```

GetLockStatusData(void)
{...
    for (i = 0; i < ProcGlobal->allProcCount; ++i)
    {
        PGPROC          *proc = &ProcGlobal->allProcs[i];
        ...
        LWLockAcquire(&proc->backendLock, LW_SHARED);
        //在proc->backendLock锁对象上加轻量的共享锁
        ...}
    ...}
}

```

另外，PostgreSQL 中，LWLock 锁也可以在其它特定对象上加排它锁，如在 ProcArrayLock 上加锁：

```

void
analyze_rel(Oid relid, RangeVar *relation, int options,
            VacuumParams *params, List *va_cols, bool in_outer_xact,
            BufferAccessStrategy bstrategy)
{...
    // OK, let's do it. First let other backends know I'm in ANALYZE.
    // ProcArrayLock宏定义，进程数组锁： #define ProcArrayLock (&MainLWLockArray[4].lock)
    LWLockAcquire(ProcArrayLock, LW_EXCLUSIVE); //在ProcArrayLock锁对象上加轻量的排它锁
    MyPgXact->vacuumFlags |= PROC_IN_ANALYZE;
    LWLockRelease(ProcArrayLock); //快速释放锁
    ...}
}

```

在 buf 上加锁或释放锁：

```

LockBuffer(Buffer buffer, int mode) //在buf上加锁或释放锁
{...
    buf = GetBufferDescriptor(buffer - 1); //获取buf
}

```



```

if (mode == BUFFER_LOCK_UNLOCK)
    LWLockRelease(BufferDescriptorGetContentLock(buf)); //在buf上释放锁
else if (mode == BUFFER_LOCK_SHARE)
    LWLockAcquire(BufferDescriptorGetContentLock(buf), LW_SHARED); //在buf上加锁
else if (mode == BUFFER_LOCK_EXCLUSIVE)
    LWLockAcquire(BufferDescriptorGetContentLock(buf), LW_EXCLUSIVE); //在buf上加锁
else
    elog(ERROR, "unrecognized buffer lock mode: %d", mode);
}

```

## 8. 缓存区的封锁管理

共享缓存区的读写,存在竞争操作,因而需要使用锁进行保护。PostgreSQL 使用 LWLock 锁的来保护共享缓存区,但是对共享缓存区采取分段保护的策略,即把共享缓存区分为几个子块,每个子块使用一个 LWLock 锁进行保护,这样做的好处,能增大对共享缓存区的并发访问粒度、减少竞争冲突。

```

/*
 * The shared buffer mapping table is partitioned to reduce contention.
 * To determine which partition lock a given tag requires, compute the tag's
 * hash code with BufTableHashCode(), then apply BufMappingPartitionLock().
 * NB: NUM_BUFFER_PARTITIONS must be a power of 2!
 */
#define BufTableHashPartition(hashcode) \
    ((hashcode) % NUM_BUFFER_PARTITIONS)
#define BufMappingPartitionLock(hashcode) \
    (&MainLWLockArray[BUFFER_MAPPING_LWLOCK_OFFSET + \
        BufTableHashPartition(hashcode)].lock)
#define BufMappingPartitionLockByIndex(i) \
    (&MainLWLockArray[BUFFER_MAPPING_LWLOCK_OFFSET + (i)].lock)

```

同理,锁管理器的共享的 Hash 表 (LockHashPartitionLock)、谓词锁的共享 Hash 表 (PredicateLockHashPartitionLock),也被分段后用不同的 LWLock 加以保护。

### 8.2.3 SpinLock 与 LWLock 比较

从系统的生命周期看,这两种锁因加锁对象都是系统层级所以都是在系统初始化期间建立的。但是从使用的范围和粒度看却不同:SpinLock 锁的粒度非常小,通常保护的都是结构体,是对“自我”的保护;LWLock 锁的粒度相对大一些,通常保护的是内存结构一级的对象,如一个 hash 表,这个属于“系统业务”级别的保护。在下面这个示例中,我们可以体会这个差别。

```

typedef struct pgssSharedState
//本结构体中的两种锁的详细用法,可以参见: pg_stat_statements_internal()

```

```

{
    LWLock      *lock;                /* protects hashtable search/modification */
    //用以保护 “static HTAB *pgss_hash”
    double      cur_median_usage;     /* current median usage in hashtable */
    Size        mean_query_len;       /* current mean entry text length */
    slock_t     mutex;                /* protects following fields only: */
    //用以保护pgssSharedState这个结构体自身的数据
    Size        extent;               /* current extent of query file */
    int         n_writers;             /* number of active writers to query file */
    int         gc_count;              /* query file garbage collection cycle count */
} pgssSharedState;

```

## 8.3 事务锁

对于事务类型的锁，PostgreSQL 提供了支持，范围包括元数据之间的并发保护，也包括用户数据之间的并发保护。但是，PostgreSQL 没有像 MySQL 一样，从概念上明确区分元数据锁（MDL\_lck）和行级记录锁（lock\_rec\_t），而是把他们作为一个整体如定义为了 RegularLock 锁，加以处理。

但是，再怎么融合完好的系统，都需要从逻辑上区分出元数据锁和元组级的锁。

PostgreSQL 提供事务锁 RegularLock 锁的管理，事务锁中包括了元数据类型的锁和元组级的锁，没有像 InnoDB 把元数据锁（DDL 操作）和记录锁（类似元组级锁但不是）分为两类进行管理。

PostgreSQL 元组级的锁不是在事务完成时释放的，而是某个 DML 操作完成即释放，这点也不与 InnoDB 相同。

PostgreSQL 元数据类型的锁，是在事务完成时释放，这点符合 SS2PL，与 InnoDB 保持一致。

PostgreSQL 也提供死锁检测。但是死锁检测同时检测元数据类型的锁和元组级的锁。而 InnoDB 分别对元数据锁和记录锁各自进行死锁检测。这是因为两者在元组一级施加锁的对象是不一样的，PostgreSQL 施加锁的对象是元组，元组有多个版本，在不同的版本上进行操作避免了并发带来的问题。而 InnoDB 的元组虽然也有多个版本，但是锁施加的对象是索引项上的记录不是元组，所以不能利用多版本带来的好处。

### 8.3.1 锁的基本信息

#### 1. 锁的粒度

在加锁的命令中，支持八种加锁的粒度，数据内部对应为 “typedef int LOCKMODE”，使用方式如表 8-7 所示。



```

/* NoLock is not a lock mode, but a flag value meaning "don't get a lock" */
#define NoLock 0
#define AccessShareLock 1 /* SELECT命令可以在表上获得本锁 */
#define RowShareLock 2 /* SELECT FOR UPDATE/FOR SHARE命令可以在表上获得本锁 */
#define RowExclusiveLock 3 /* INSERT, UPDATE, DELETE 命令可以在表上获得本锁 */
#define ShareUpdateExclusiveLock 4 /* VACUUM (non-FULL), ANALYZE, CREATE INDEX CONCURRENTLY命令可以在表上获得本锁 */
#define ShareLock 5 /* CREATE INDEX (WITHOUT CONCURRENTLY) 命令可以在表上获得本锁 */
#define ShareRowExclusiveLock 6 /* like EXCLUSIVE MODE, but allows ROW SHARE, 只有一个会话可以持有此锁, 这样的会话中获得本锁是由于执行了CREATE TRIGGER、ALTER TABLE的部分语句 (ALTER TABLE启用、禁止触发器和约束) */
#define ExclusiveLock 7 /* blocks ROW SHARE/SELECT...FOR UPDATE, 只允许另外的事务读被本事务锁住的表, 即其他事务可以加AccessShareLock锁。许多情形下, 可以获取本锁 */
#define AccessExclusiveLock 8 /* ALTER TABLE, DROP TABLE, VACUUM FULL, and unqualified LOCK TABLE等需要完全锁定表对象的命令, 都申请本锁以排斥其他任何在此表上的操作, 再比如TRUNCATE操作 */

```

表 8-7 锁的粒度使用说明表

锁	用法
AccessShareLock	主要用于对系统表进行读取数据时加锁（读锁），如创建一个表元数据需要写入 pg_class，则要对 pg_class 加排它锁；
RowShareLock	主要用于对用户表进行查询时因 FOR UPDATE/FOR SHARE 操作而加锁（读锁）
RowExclusiveLock	主要用于对系统表进行修改时加锁（写锁），如创建一个表元数据需要写入 pg_class，则要对 pg_class 加排它锁； 另外对于用户表，执行 INSERT、UPDATE、DELETE 操作时使用
ShareUpdateExclusiveLock	主要用于对系统表 pg_class 等加排它锁，执行 VACUUM（没有 FULL 选项）、ANALYZE、CREATE INDEX CONCURRENTLY、BRIN indexes <sup>①</sup> 操作时施加
ShareLock	主要用于创建索引时加锁（写锁），但存在特殊用法：等待指定的事务提交 / 回滚操作完成
ShareRowExclusiveLock	在用户表上为操作触发器和约束加排它锁
ExclusiveLock	排它锁，比如用于 ROW SHARE/SELECT...FOR UPDATE，但不仅限于此，比如执行 VACUUM、存储层物理页面的扩展等。另外，不仅限于事务操作使用，系统锁也可以使用，如 XactLockTableInsert() 在分配事务 ID 时等很多地方会被调用
AccessExclusiveLock	主要用于对系统表进行修改时加锁（排它锁），最强级别的锁，没有新的锁可以再升级（即任何锁都被排斥）

① 9.5 版本提供的块索引，可以加快大量数据的范围扫描速度。BRIN indexes intend to enable very fast scanning of extremely large tables。

## 2. 锁对象

每一个对象被施加的锁，使用 Lock 来存储相关的锁信息，如锁标志、授予信息、等待信息等。

一个数据库对象，诸如元组对象，此对象上只有一个锁存在，即有一个 LOCK 对象。



在此 LOCK 对象上, 存有此数据库对象上所申请、授予的各种类型的锁、次数等信息。

```

/*
 * Per-locked-object lock information: //如下注释, 应仔细研读理解
 *
 * tag          -- uniquely identifies the object being locked
 * grantMask    -- bitmask for all lock types currently granted on this object.
 * waitMask     -- bitmask for all lock types currently awaited on this object.
 * procLocks    -- list of PROCLOCK objects for this lock.
 * waitProcs    -- queue of processes waiting for this lock.
 * requested    -- count of each lock type currently requested on the lock (includes
requests already granted!!).
 * nRequested   -- total requested locks of all types.
 * granted      -- count of each lock type currently granted on the lock.
 * nGranted     -- total granted locks of all types.
 *
 * Note: these counts count 1 for each backend. Internally to a backend,
 * there may be multiple grabs on a particular lock, but this is not reflected
 * into shared memory.
 */
typedef struct LOCK //锁有其属主, 属主必定是数据库的一个对象, 如表对象、行数据等
{
    /* hash key */
    LOCKTAG    tag;          /* 锁对象的唯一的标志, 如是行级锁、元数据锁之Relation锁等。
是Hash类锁表的快速查找键 */
    /* data */
    LOCKMASK    grantMask;   /* 已经被授予的锁类型构成的位图, 锁类型指的是LOCKMODE定义的
值如 "AccessExclusiveLock", 即上一节的锁粒度
    LOCKMASK    waitMask;    /* 正在等待的锁构成的位图, 即锁请求 */
    SHM_QUEUE    procLocks;  /* list of PROCLOCK objects assoc. with lock */
    /*持有本锁对象的所有的会话进程列表
    PROC_QUEUE    waitProcs; /* list of PGPROC objects waiting on lock */
    /*等待本锁对象的所有的会话进程列表
    int            requested[MAX_LOCKMODES]; /* counts of requested locks */
    /*申请本锁对象的不锁模式的次数
    int            nRequested;                /* total of requested[] array */
    /*所有申请本锁对象的不锁模式的次数, 上一个数组的总和
    int            granted[MAX_LOCKMODES];   /* counts of granted locks */
    /*授予本锁对象的不锁模式的次数
    int            nGranted;                /* total of granted[] array */
    /*所有授予本对象的不锁模式的次数, 上一个数组的总和
} LOCK;

```

一个锁对象的标签(锁标志)定义如下:

```

typedef struct LOCKTAG //一个四元组, 如果是一个元组锁, 则四元组为{dbid, relid,
    blocknum, offset}, 页锁为{dbid, relid, blocknum, NULL}

```

```

{
    //关系锁为{dbid, relid, NULL, NULL}, 此四元组对应本结构体的头四个元素
    uint32    locktag_field1; /* a 32-bit ID field */      //dbid
    uint32    locktag_field2; /* a 32-bit ID field */      //relbid
    uint32    locktag_field3; /* a 32-bit ID field */      //blocknum
    uint16    locktag_field4; /* a 16-bit ID field */      //offset
    uint8     locktag_type; /* see enum LockTagType */    //锁的类型, 如关系锁、
元组锁、事务上的锁等, 参见8.3.2节RegularLock锁
    uint8     locktag_lockmethodid; /* lockmethod indicator */ //锁方法的id
} LOCKTAG;

```

### 3. 锁表

锁对象, 会被注册在锁表中, 便于查找。这就是锁表, 是锁存放的地方 (实际上是锁的地址聚集存放之处)。

而对于同一个对象的竞争, 来自不同的会话进程, 所以锁表对象, 是全局的。为了加快锁的查找, PostgreSQL 提供了两种机制, 一是启用了会话进程的本地锁表, 把一些只有自己关注的 (没有产生竞争) 的锁, 存放在本地锁表, 便于快速使用; 二是为一些冲突稀少的锁提供快速的访问方式 (实际上冲突并不稀少), 如对于 SELECT、INSERT、UPDATE 和 DELETE 操作引发的 AccessShareLock、RowShareLock、RowExclusiveLock 类的锁, 定义为 “Weak relation locks”, 把锁注册到进程结构中以备快速查找 (参见 8.2.2 节对于进程结构体的说明和 8.4.1 节对于获取锁的说明), 目的是减少锁的施加和释放等操作的代价。只有以上两种方式不能定位锁时, 才从全局的锁表中继续进行查找 (申请锁时, 查找不到则会创建, 参见 8.4.2 节)。

另外, 全局锁表 LockMethodLockHash 中存放的信息是某个对象上的丰富的申请锁、授予锁等信息, 其元素对应的是 “LOCK”。而本地锁表 LockMethodLocalHash 中存放的信息不仅包括了前句所述的锁信息, 还包括了关联进程和有关锁的整体情况如锁资源属主等更多信息, 其元素对应的是 “LOCALLOCK”。可以简单认为, 本地锁表相关于全局锁表和全局用户会话进程表的总和 (不精确, 只是用以表包含意, 本地包含全局的于本会话进程相关的锁信息)。

```

/* The LockMethodLockHash and LockMethodProcLockHash hash tables are in
 * shared memory; LockMethodLocalHash is local to each backend.
 */
static HTAB *LockMethodLockHash;          //位于共享缓存中, 全局变量, 表示全局锁表与锁方法
                                           之间的映射
static HTAB *LockMethodProcLockHash;      //位于共享缓存中, 全局变量, 表示持锁者与锁方法之
                                           间的映射
static HTAB *LockMethodLocalHash;         //表示本会话内的局部锁表, 如果本地锁表中不存在锁
                                           对象, 才从全局锁表中查找

```

“Weak relation locks” 类锁的定义如下, 注意其与一个进程结构体 “struct PGPROC” 的 fpRelId 和 fpLockBits 配合使用, 可参阅 FastPathGrantRelationLock() 函数理解 Postgre-



SQL 定义的锁快速访问方法。锁快速访问方法定义结构体如下：

```
typedef struct
{
    slock_t      mutex;
    uint32       count[FAST_PATH_STRONG_LOCK_HASH_PARTITIONS];
} FastPathStrongRelationLockData;

static volatile FastPathStrongRelationLockData *FastPathStrongRelationLocks;
```

#### 4. 加锁对象

加锁对象不仅包括了锁对象，而且还包括了加锁的模式，这二者合起来构成了完整的“锁信息”，表示给什么对象加什么样的锁，在使用的时候，作为加锁请求的标志（注意加锁请求的标志和锁标志是两个不同的结构体，有着不同的含义）。

```
typedef struct LOCALLOCKTAG    //锁对象上加什么粒度的锁
{
    LOCKTAG      lock;    /* identifies the lockable object */ //锁对象上
    LOCKMODE     mode;    /* lock mode for this table entry */ //加什么粒度的锁
} LOCALLOCKTAG;
```

#### 5. 局部锁表

每个会话进程持有的锁如果没有竞争发生，则保存在本地的锁表中。

```
typedef struct LOCALLOCK
{
    LOCALLOCKTAG tag;    /* unique identifier of locallock entry */
    //加锁请求的标志

    /* data */
    LOCK      *lock;    /* associated LOCK object, if any */
    //加锁请求所申请到的锁
    PROCLOCK  *proclock;    /* associated PROCLOCK object, if any */
    //持锁者是哪个会话进程
    uint32     hashcode;    /* copy of LOCKTAG's hash value */
    //锁标志的hash值
    int64      nLocks;    /* total number of times lock is held */
    //锁被持有的次数
    int        numLockOwners;    /* # of relevant ResourceOwners */
    //与本锁对象相关的ResourceOwner有多少
    int        maxLockOwners;    /* allocated size of array */
    bool       holdsStrongLockCount;    /* bumped FastPathStrongRelationLocks */
    LOCALLOCKOWNER *lockOwners;    /* dynamically resizable array */
} LOCALLOCK;
```



## 6. 持锁者

锁被谁持有？每个锁对象总是有一个持锁者，持锁者根据 SQL 语句的操作语义确定请在什么对象上加什么样的锁（LOCALLOCKTAG），加锁申请成功则确定把锁归属于谁（锁隶属于哪个会话进程）。

```
typedef struct PROCLockTAG
//标识会话进程和锁的关系，一个会话进程会持有很多锁，而一个锁只能被一个会话进程持有
{
    /* NB: we assume this struct contains no padding! */
    LOCK      *myLock;      /* link to per-lockable-object information */ //标识锁
    PGPROC     *myProc;     /* link to PGPROC of owning backend */ //标识会话进程
} PROCLockTAG;

typedef struct PROCLock      //标识会话进程和锁的关联关系
{
    /* tag */
    PROCLockTAG tag; /* unique identifier of proclock object */ //标识会话进程与锁

    /* data */
    PGPROC      *groupLeader; //为完成同一个任务而并行的进程属于同一个组，持锁被视为同
    一个事务
    LOCKMASK     holdMask;    /* bitmask for lock types currently held */
    //当前持有锁者持有的锁类型
    LOCKMASK     releaseMask; /* bitmask for lock types to be released */
    //当前持有锁者释放的锁类型
    SHM_QUEUE    lockLink;    /* list link in LOCK's list of proclocks */
    //本对象PROCLockTAG在锁的链表中的位置，参见“锁对象”一节
    SHM_QUEUE    procLink;    /* list link in PGPROC's list of proclocks */
    //本对象PROCLockTAG在进程链表中的位置⊖
} PROCLock;
```

## 7. 锁相容性表和锁方法表

PostgreSQL 使用一个 INT 型数组巧妙的定义了锁之间的相容性列表，用以检查已经持有的锁是否排斥新申请的锁，方式如下：

```
static const LOCKMASK LockConflicts[] = {
    0, //值为零，表示空一行，下一行才是真的锁模式
    //检查与AccessExclusiveLock锁的相容性
    LOCKBIT_ON(AccessExclusiveLock), //对于AccessShareLock类型的持锁，2左移8位是256，
    其二进制为“100000000”，后八位是零，对于8种可能申请的锁类型，与零匹配之后还是零，表示不
    相容。如下依次计算方式类推，可以知道新申请的锁与持有的锁是否相容
```

⊖ 在一个进程结构体中，定义有“SHM\_QUEUE myProcLocks[NUM\_LOCK\_PARTITIONS];”，一个 PROCLock对象列表，即所有持有锁的进程的PROCLock对象列表。



```

//检查与RowShareLock锁的相容性
LOCKBIT_ON(ExclusiveLock) | LOCKBIT_ON(AccessExclusiveLock),
//检查与RowShareLock锁的相容性

//检查与RowExclusiveLock锁的相容性
LOCKBIT_ON(ShareLock) | LOCKBIT_ON(ShareRowExclusiveLock) | LOCKBIT_
ON(ExclusiveLock) | LOCKBIT_ON(AccessExclusiveLock),

//检查与ShareUpdateExclusiveLock锁的相容性
LOCKBIT_ON(ShareUpdateExclusiveLock) |
LOCKBIT_ON(ShareLock) | LOCKBIT_ON(ShareRowExclusiveLock) | LOCKBIT_
ON(ExclusiveLock) | LOCKBIT_ON(AccessExclusiveLock),

//检查与ShareLock锁的相容性
LOCKBIT_ON(RowExclusiveLock) | LOCKBIT_ON(ShareUpdateExclusiveLock) |
LOCKBIT_ON(ShareRowExclusiveLock) | LOCKBIT_ON(ExclusiveLock) | LOCKBIT_
ON(AccessExclusiveLock),

//检查与ShareRowExclusiveLock锁的相容性
LOCKBIT_ON(RowExclusiveLock) | LOCKBIT_ON(ShareUpdateExclusiveLock) |
LOCKBIT_ON(ShareLock) | LOCKBIT_ON(ShareRowExclusiveLock) | LOCKBIT_
ON(ExclusiveLock) | LOCKBIT_ON(AccessExclusiveLock),

//检查与ExclusiveLock锁的相容性
LOCKBIT_ON(RowShareLock) | LOCKBIT_ON(RowExclusiveLock) | LOCKBIT_
ON(ShareUpdateExclusiveLock) |
LOCKBIT_ON(ShareLock) | LOCKBIT_ON(ShareRowExclusiveLock) | LOCKBIT_
ON(ExclusiveLock) | LOCKBIT_ON(AccessExclusiveLock),

//检查与AccessExclusiveLock锁的相容性
LOCKBIT_ON(AccessShareLock) | LOCKBIT_ON(RowShareLock) | LOCKBIT_
ON(RowExclusiveLock) | LOCKBIT_ON(ShareUpdateExclusiveLock) |
LOCKBIT_ON(ShareLock) | LOCKBIT_ON(ShareRowExclusiveLock) | LOCKBIT_
ON(ExclusiveLock) | LOCKBIT_ON(AccessExclusiveLock)
};

```

基于锁相容性表，又定义了锁的方法表，如下：

```

typedef struct LockMethodData //锁方法表
{
    int numLockModes; //一个锁方法表里，锁模式的个数。系统自定义了两个锁方法表，都是8
    个锁模式（采用宏AccessExclusiveLock的值）
    const LOCKMASK *conflictTab; //锁的相容性列表，对应了“default_lockmethod”
    和“user_lockmethod”中都使用了的“LockConflicts”
    const char *const * lockModeNames; //锁方法名称
    const bool *trace_flag; //用于调试目的的跟踪信息
} LockMethodData;

typedef const LockMethodData *LockMethod;

```

锁方法数据结构就是一个锁表，此锁表被用于定义锁方法表数组，此数组中定义了两个锁方法表的 ID，取值可以为 1(为缺省使用的方法，DEFAULT\_LOCKMETHOD) 和 2(用户自定义的方法，USER\_LOCKMETHOD)，这两个锁方法表，第一种是被 PostgreSQL 作为缺省的系统默认使用的锁相容性判断方法，另外一种为用户定义的（特指劝告锁，参见 8.6.6 节），但不是用户可自定义的。

### 8.3.2 ReguarLock

ReguarLock 锁，用于对 PostgreSQL 数据库系统的元数据和用户数据进行封锁管理。

这样的锁，通过 LockTagType 定义有 10 种类型，分别用于不同的封锁场景；这些锁既可以使用到系统默认的封锁方式中（自动加锁），也可以显式地用于用户编写的 SQL 语句中（用户主动加锁）。显式的方式加锁是通过在 SQL 语句中直接指定加锁的粒度来完成加锁操作的，系统自动加锁是数据库系统根据并发管理的需要而自动完成加锁操作；不管是自动加锁还是用户显式加锁，锁将随着事务的结束而被自动释放（session 级的劝告锁除外）。

锁主要是施加在并发时可能操作的数据对象上，这样的对象可能是“RELATION”“PAGE”“TUPLE”和任何“OBJECT”，另外还包含一些特例如“TRANSACTION”“VIRTUAL TRANSACTION”“USER”等，只要存在被并发操作的可能，都可以通过锁来抑制并发进而保护“被保护的對象”在一个事务内“数据是一致的”。

“数据是一致的”指的是数据在事务管理系统操作之下，前后逻辑保持一致，没有被其他并发的事务修改致使这个数据的“前后逻辑不能保持一致”。这样说起来，似乎有点拗口，换句话说，就是数据不会发生一些“事务异常”现象，比如说，不会发生 ANSI 的 SQL 标准规定的三类异常（脏读、不可重复读、幻读），或者不会发生写偏序一类的异常。

PostgreSQL V9.1 之前的版本会有写偏序的异常，原因是早期的版本实现的 Snapshot Isolation 只能屏蔽脏读、不可重复读、幻读三种异常，而没有做到 Serializable，V9.1 的版本通过可串行化快照隔离（Serializable Snapshot Isolation，SSI）技术，才避免了写偏序的问题，保证了数据的一致性。

#### 1. RegularLock 锁的类型

RegularLock 锁（常规锁）用于事务间并发保护用户表的数据，定义的 10 种类型如下：

```
/* LOCKTAG is the key information needed to look up a LOCK item in the lock hashtable.
 * A LOCKTAG value uniquely identifies a lockable object.
 *
 * The LockTagType enum defines the different kinds of objects we can lock. We can
 * handle up to 256 different LockTagTypes.
 */
typedef enum LockTagType //枚举了可以加锁的对象，同时也能表明锁的类型。各个具体的枚举值的
更详细说明，参见表 8-8
{
```



```
LOCKTAG_RELATION, /* whole relation */ //在关系（表、视图、索引等）上加
“RELATION”锁，主要用于CREATE、DROP、TRUNCATE、VACUUM等操作。可见这是一个元数据类型的锁，
类似MySQL的MDL_lock锁
/* ID info for a relation is DB OID + REL OID; DB OID = 0 if shared */
LOCKTAG_RELATION_EXTEND, /* the right to extend a relation */
//物理文件因存储空间不够做扩展，如表的数据的增加会导致文件变大
/* same ID info as RELATION */
LOCKTAG_PAGE, /* one page of a relation */
//在物理页面上加“PAGE”锁，主要用于索引相关的INSERT、DELETE操作
/* ID info for a page is RELATION info + BlockNumber */
LOCKTAG_TUPLE, /* one physical tuple */
//在物理元组上加“TUPLE”锁，主要用于INSERT、UPDATE操作，实际是行级锁
/* ID info for a tuple is PAGE info + OffsetNumber */
LOCKTAG_TRANSACTION, /* transaction (for waiting for xact done) */
//事务的锁
/* ID info for a transaction is its TransactionId */
LOCKTAG_VIRTUALTRANSACTION, /* virtual transaction (ditto) */
//虚拟事务的锁
/* ID info for a virtual transaction is its VirtualTransactionId */
LOCKTAG_SPECULATIVE_TOKEN, /* speculative insertion Xid and token */
//事务的锁
/* ID info for a transaction is its TransactionId */
LOCKTAG_OBJECT, /* non-relation database object */
//非“RELATION”锁，如database
/* ID info for an object is DB OID + CLASS OID + OBJECT OID + SUBID */

/* Note: object ID has same representation as in pg_depend and pg_description,
 * but notice that we are constraining SUBID to 16 bits.
 * Also, we use DB OID = 0 for shared objects such as tablespaces. */
LOCKTAG_USERLOCK, /* reserved for old contrib/userlock code */
//用户锁
LOCKTAG_ADVISORY /* advisory user locks */ //劝告锁，供用户显式加锁使用
} LockTagType;
```

PostgreSQL 数据级的锁，主要调用关系如图 8-2 ~ 图 8-8 所示。

表 8-8 锁标识说明表

锁标识的名称	锁的主要用途	使用方式
LOCKTAG_RELATION	在关系（表、视图、索引等）上加“RELATION”锁，主要用于CREATE、DROP、TRUNCATE、VACUUM等操作，也被用于“standby” <sup>①</sup> 系统，如图 8-2 所示	自动封锁 + 主动封锁
LOCKTAG_RELATION_EXTEND	物理文件因存储空间不够做扩展，如表的数据的增加会导致文件变大，预先分配的空间不足，则文件自动扩展，这时需要对文件加锁防止并行操作带来的物理存储层面的不一致	自动封锁

(续)

锁标识的名称	锁的主要用途	使用方式
LOCKTAG_PAGE	在物理页面上加“PAGE”锁，主要用于索引相关的 INSERT、DELETE 操作，如图 8-3 所示	自动封锁 + 主动封锁
LOCKTAG_TUPLE	在物理元组上加“TUPLE”锁，主要用于 DELETE、UPDATE 操作，如图 8-4 所示	自动封锁 + 主动封锁
LOCKTAG_TRANSACTION	事务锁，主要用于事务号的分配、子事务提交时使用，是事务管理对数据级操作时的辅助事务实现数据一致性的措施，如图 8-5 所示	自动封锁
LOCKTAG_VIRTUALTRANSACTION	虚拟事务锁，如图 8-6 所示	自动封锁
LOCKTAG_SPECULATIVE_TOKEN	插入操作的相关锁，如图 8-7 所示	自动封锁
LOCKTAG_OBJECT	对象锁，如图 8-8 所示	自动封锁
LOCKTAG_USERLOCK	用户锁	自动封锁
LOCKTAG_ADVISORY	劝告锁，供用户显式加锁使用	主动封锁

- ① PostgreSQL 中称为 hot standby，类似 Oracle 的 active dataguard，使得从数据库在应用 WAL 日志的同时，可以提供只读服务

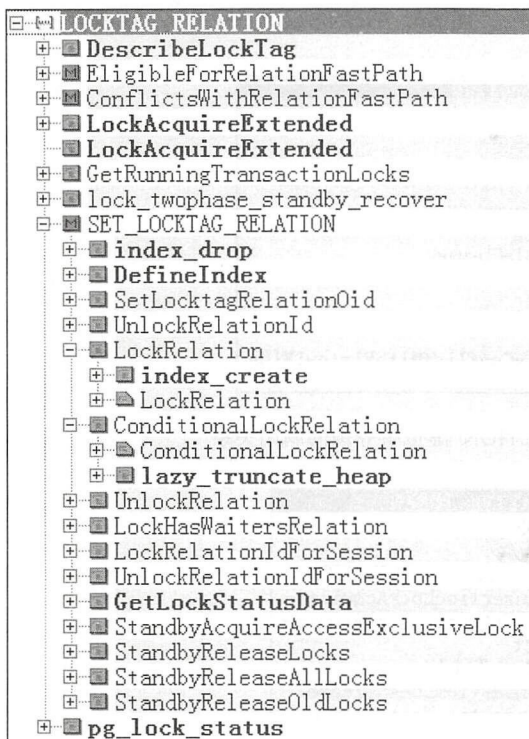


图 8-2 RELATION/ 关系锁的调用关系



图 8-3 PAGE/ 页锁的调用关系



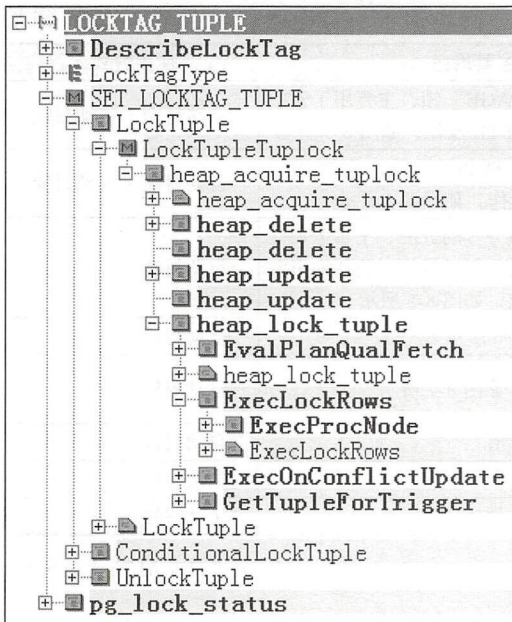


图 8-4 TUPLE/元组锁的调用关系

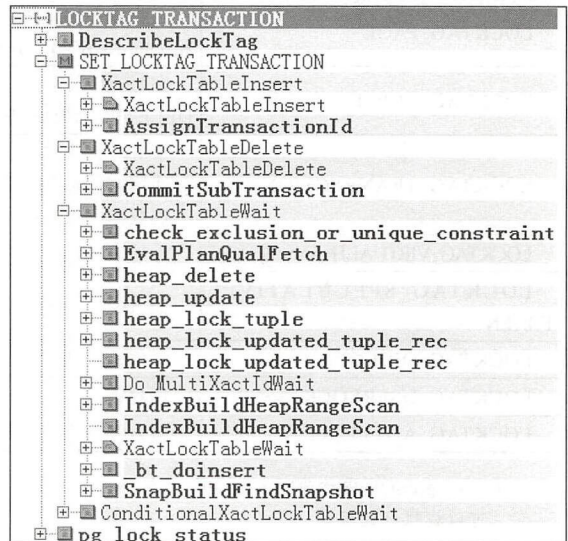


图 8-5 TRANSACTION/事务锁的调用关系

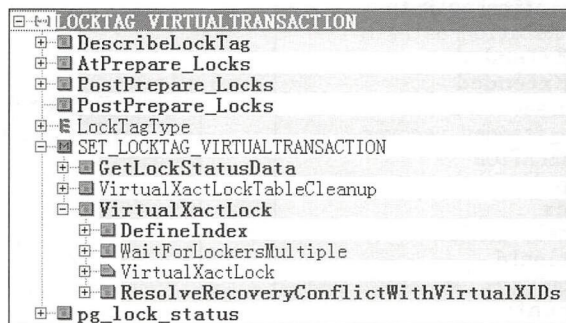


图 8-6 VIRTUAL TRANSACTION/虚拟事务锁的调用关系

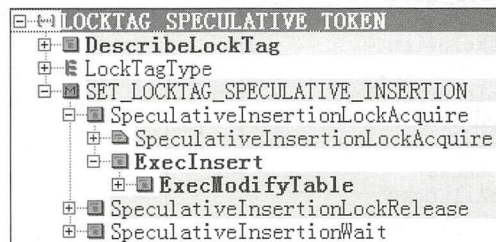


图 8-7 INSERT 操作事务锁的调用关系





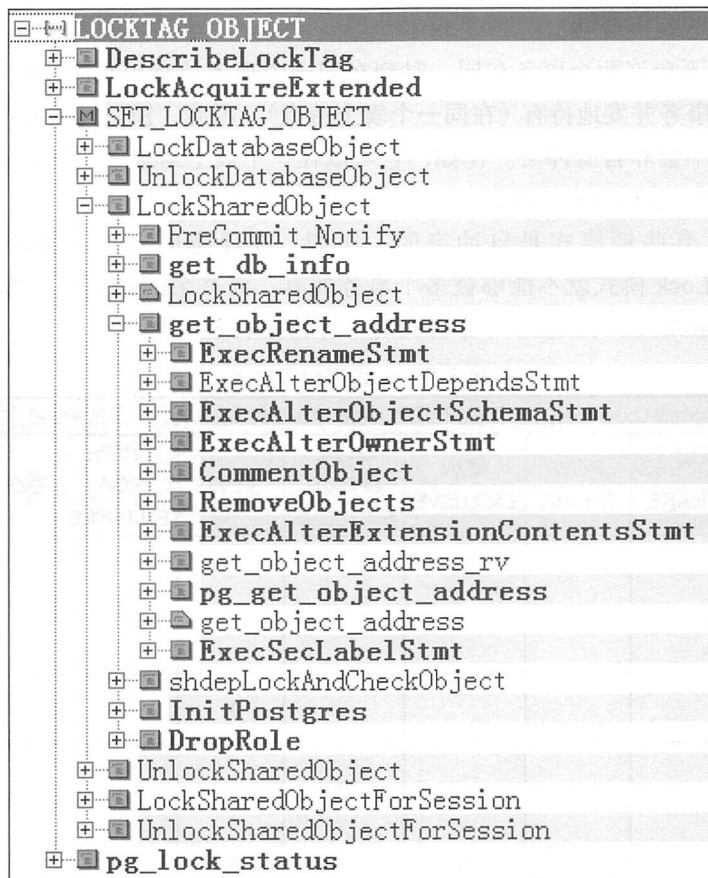


图 8-8 OBJECT/ 对象锁的调用关系

## 2. 表级显式锁

PostgreSQL 提供显式地对表对象加锁操作，所加的锁在事务结束时（commit 或 rollback）被释放。

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
where lockmode is one of:
```

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

注意：有些锁的名称中带有“Row”，这并不表明此锁是行级锁，PostgreSQL 官方文档说明如下：

Remember that all of these lock modes are table-level locks, even if the name contains the word "row"; the names of the lock modes are historical.



3. 表级锁之间的相容性

PostgreSQL 不允许两个事务在同一时刻在同一个表上持有相互冲突的锁。但非冲突锁模式可以由许多事务并发地持有（在同一个表上持有）。因此，锁之间需要考虑是否相容。

一个事务决不会和自身冲突。比如，它可以在一个表上请求 ACCESS EXCLUSIVE 然后稍后的时候请求 ACCESS SHARE。

个别情况，有些锁模式是自冲突的；比如，在任意时刻 ShareRowExclusiveLock、AccessExclusiveLock 模式就不能够被多个事务拥有。而其余的锁模式，都不是自冲突的；比如，AccessShareLock 可以被多个事务持有。详情如表 8-9 所示。

表 8-9 表级锁相容性表

Requested Lock Mode	Current Lock Mode(X 表示准备申请的锁被已经加锁的事务排斥，即不允许加锁。空白表示允许加锁)							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X		X	X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

4. 表级锁加锁和释放锁

表级锁加锁的请求，是通过 LockAcquire() 函数完成。

```
void
LockRelation(Relation relation, LOCKMODE lockmode)
{...
    SET_LOCKTAG_RELATION(tag, //设置关系的标志
        relation->rd_lockInfo.lockRelId.dbId, relation->rd_lockInfo.lockRelId.relId);
    res = LockAcquire(&tag, lockmode, false, false); //然后通过“LockAcquire”完成加锁操作
    ...
}
```



表级锁释放锁的请求，是通过 LockRelease() 函数完成。

```
void
UnlockRelation(Relation relation, LOCKMODE lockmode)
{...
    SET_LOCKTAG_RELATION(tag, //设置关系的标志
                          relation->rd_lockInfo.lockRelId.dbId, relation->rd_lockInfo.lockRelId.relId);

    LockRelease(&tag, lockmode, false); //然后通过“LockRelease”完成释放锁操作
}
```

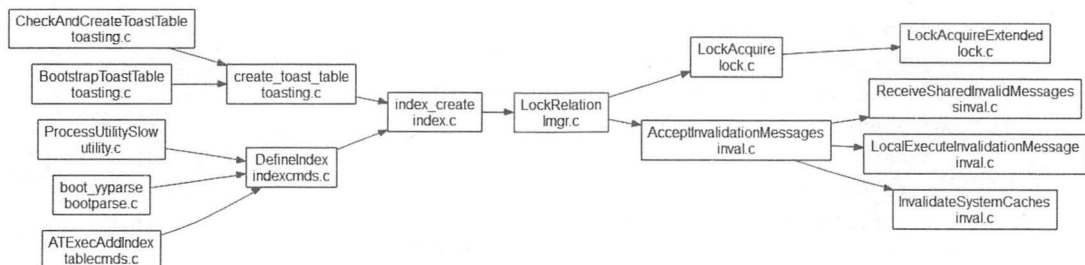


图 8-9 表级锁上下文调用关系

从图 8-9 可以看出，创建索引的时候，才会调用 LockRelation() 加锁。但是，这似乎不可能，读写表的数据，都应该在表级加锁，那为什么只有 index\_create() 函数调用了 LockRelation() 函数呢？

PostgreSQL 对于 LockRelation() 函数的解释如下：

```
/* This is a convenience routine for acquiring an additional lock on an
 * already-open relation. Never try to do "relation_open(foo, NoLock)"
 * and then lock with this. */
```

这表明，LockRelation() 函数只是用于在一个已经打开的表（表述为 RELATION 更准确）上，获取一个额外的锁。真正的表级加锁过程，参见下一节。

## 5. 表级锁加锁的过程——查询语句

以“SELECT \* FROM parent;”为例，观察加锁过程与调用的函数栈。

首先，在 SQL 的分析阶段，即调用 relation\_open() 函数但不调用 LockRelationOid() 函数，此时不对表加锁。

```
PostgresMain()
exec_simple_query() //执行一个简单的查询语句
pg_analyze_and_rewrite()
parse_analyze() //分析阶段
transformTopLevelStmt()
transformStmt()
```





```

transformSelectStmt()
transformFromClause() //分析FROM子句
transformFromClauseItem()
transformTableEntry() //分析FROM子句的表/视图等对象
parserOpenTable()
heap_openrv_extended()
relation_openrv_extended()
relation_open() //分析阶段即打开表，后续要读入表的列对象等，
                尚不涉及事务，所以锁值为0

```

其次，在分析阶段，当要获取元组的元信息的时候，为表/视图加 AccessShareLock 锁。

```

PostgresMain()
exec_simple_query() //执行一个简单的查询语句
pg_analyze_and_rewrite()
parse_analyze() //分析阶段
transformTopLevelStmt()
transformStmt()
transformSelectStmt() //分析FROM子句
transformTargetList() //分析目标列子句
ExpandColumnRefStar() //把目标列中的“*”扩展为列对象
ExpandAllTables()
expandRelAttrs()
expandRTE()
expandRelation() //获取元组的描述信息，为表加AccessShareLock锁
relation_open()
                LockRelationOid(relationId, lockmode=1=AccessShareLock)
                //此时为表/视图即RELATION加AccessShareLock锁

```

第三，在优化阶段打开表但不加锁；此过程可能执行多次；

```

PostgresMain()
exec_simple_query() //执行一个简单的查询语句
pg_analyze_and_rewrite()
pg_rewrite_query() //查询重写，即逻辑优化阶段
QueryRewrite()
fireRIRrules() //对于每一个范围表应用RIR规则
heap_open()
relation_open() //此时为表/视图即RELATION不加锁

```

第四，生成执行计划时，无锁方式打开 RELATION；

```

PostgresMain()
exec_simple_query() //执行一个简单的查询语句
pg_plan_queries() //生成执行计划
pg_plan_query()
planner()
standard_planner()

```



```

subquery_planner()
grouping_planner()
query_planner()
    add_base_rels_to_query() //FROM子句
    add_base_rels_to_query() //FROM子句中的范围表
    build_simple_rel()
    get_relation_info()
    heap_open()
        relation_open() //不加锁的方式打开表/视图等，因为之前已经
                        打开过，且不做修改

```

第五，获取表上索引的信息，为此加 **AccessShareLock** 锁，原因如下：

```

/* For each index, we get the same type of lock that the executor will need, and
do not release it.
* This saves a couple of trips to the shared lock manager while not creating
any real loss of concurrency,
* because no schema changes could be happening on the index while we hold
lock on the parent rel,
* and neither lock type blocks any other kind of index operation. */
if (rel->relid == root->parse->resultRelation)
    lmode = RowExclusiveLock;
else
    lmode = AccessShareLock;

```

调用栈如下：

```

PostgresMain()
    exec_simple_query() //执行一个简单的查询语句
    pg_plan_queries()
    pg_plan_query()
    planner()
        standard_planner()
        subquery_planner()
        grouping_planner()
        query_planner()
            add_base_rels_to_query()
            add_base_rels_to_query()
            build_simple_rel()
            get_relation_info()
            index_open() //打开索引
            relation_open() //准备为索引加锁
                LockRelationOid(relationId, lockmode=1=AccessShareLock)
                //此时为表/视图即RELATION加AccessShareLock锁

```

第六，为获取约束信息在系统表上加锁：

```

conrel = heap_open(ConstraintRelationId, AccessShareLock);

```





第七，为扫描系统表上加锁：

```
irel = index_open(indexId, AccessShareLock);
```

第八，执行阶段，为 RELATION 加 AccessShareLock 锁：

```
PostgresMain()
exec_simple_query()
PortalStart()
ExecutorStart() //执行器开始执行
InitPlan()
ExecInitNode()
ExecInitSeqScan()
InitScanRelation()
ExecOpenScanRelation()
heap_open()
relation_open()
    LockRelationOid(relationId, lockmode=1=AccessShareLock)
    //此时为表/视图即RELATION加锁
```

从前面的执行过程看，在一个表/视图这样的 RELATION 对象上加锁，会贯穿在 SQL 语句的分析、优化、执行等各个阶段。

## 6. 表级锁加锁的过程——更新语句

以如下 SQL 为例，观察加锁过程与调用的函数栈。

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE parent SET b1=10 WHERE pid=2; //读取索引，执行UPDATE
```

首先，在 SQL 的分析阶段，即调用 `relation_open()` 函数但不调用 `LockRelationOid()` 函数，此时不对表加锁。调用栈类似 SELECT 语句。但请注意，更新语句这个示例，没有列出不加锁和在系统表对象上加锁的情况，因此如下过程看似比上一个示例步骤少且简单。

其次，在逻辑优化阶段，进行查询重写时，需要打开表，此时加 AccessShareLock 锁；

```
PostgresMain()
exec_simple_query()
pg_analyze_and_rewrite() //逻辑优化阶段的查询重写
pg_rewrite_query()
QueryRewrite()
fireRIRrules()
heap_open()
relation_open()
    LockRelationOid(relationId, lockmode=1=AccessShareLock)
    //此时为表/视图即RELATION加锁
```

第三，在生成执行计划的时候，要读取索引，此时因 UPDATE 语句中使用索引，在索





引上施加了 RowExclusiveLock 锁:

```
PostgresMain()
exec_simple_query()
pg_plan_queries()
pg_plan_query()
planner()
    standard_planner()
    subquery_planner() //生成执行计划
    grouping_planner()
    query_planner()
    add_base_rels_to_query()
    add_base_rels_to_query()
    build_simple_rel()
    get_relation_info()
    index_open() //打开索引
    relation_open()
        LockRelationOid(relationId, lockmode=3=RowExclusiveLock)
        //此时为表/视图即RELATION加锁
```

第四, 执行阶段, 在关系山上施加了 RowExclusiveLock 锁:

```
PostgresMain()
exec_simple_query()
PortalRun()
PortalRunMulti()
ProcessQuery()
    ExecutorStart() //执行器开始执行
    standard_ExecutorStart()
    InitPlan()
    heap_open()
    relation_open()
        LockRelationOid(relationId, lockmode=3=RowExclusiveLock)
        //此时为表/视图即RELATION加锁
```

## 7. 页级锁

页面级别的锁, 简称页锁, 用于控制对共享缓冲池中索引的页面的读/写并发访问、以及在GIN索引上执行VACUUM操作, 可用的场景较少, 有共享、排他两种锁的模式。

前面谈到的锁的类型时, 所提出的LOCKTAG\_PAGE正是页锁的标识。页锁被数据库的事务管理系统自动施加, 在事务提交或回滚阶段自动释放。

页锁的加锁和释放锁类似表锁的封锁过程, 也是调用LockAcquire()函数和LockRelease()函数完成。

```
void
LockPage(Relation relation, BlockNumber blkno, LOCKMODE lockmode) //加页锁
```



```

{...
    SET_LOCKTAG_PAGE(tag, relation->rd_lockInfo.lockRelId.dbId, relation->
rd_lockInfo.lockRelId.relId, blkno);
    (void) LockAcquire(&tag, lockmode, false, false);
}

UnlockPage(Relation relation, BlockNumber blkno, LOCKMODE lockmode) //释放页锁
{...
    SET_LOCKTAG_PAGE(tag, relation->rd_lockInfo.lockRelId.dbId, relation->
rd_lockInfo.lockRelId.relId, blkno);
    LockRelease(&tag, lockmode, false);
}

```

### 8.3.3 行级锁

#### 1. 行级显式锁

PostgreSQL 提供显式地对元组对象加锁，所加的锁在事务结束时（commit 或 rollback）被释放。

```

SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ] //执行SELECT语句的时候，可
以指定“FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }”为元组显式加锁
    * | expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ]
    [ [ NOWAIT ] [...] ] ]

```

“FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }”在代码中对应的内容如下：

```

typedef enum LockTupleMode //元组级锁的模式
{
    LockTupleKeyShare, /* SELECT FOR KEY SHARE */
    LockTupleShare, /* SELECT FOR SHARE */
    LockTupleNoKeyExclusive, /* SELECT FOR NO KEY UPDATE, and UPDATES that don't
modify key columns */ //UPDATE操作只修改非索引列
    LockTupleExclusive /* SELECT FOR UPDATE, UPDATES that modify key

```





```
columns, and DELETE */ //UPDATE操作修改索引列
} LockTupleMode;
```

PostgreSQL 允许同一事务中的多个不同子事务在同一行持有相互冲突的锁；但不允许不同事务在同一行持有相互冲突的锁。

行级锁因 MVCC 机制，不排斥对数据的查询（查询阶段不加锁），只排斥写（系统自动加锁）或锁同一行（用户主动加锁）的操作。当一个事务持有共享锁时，其他事务申请的更新、删除或排他锁都不被允许施加。所以行级锁的作用就是禁止并发修改。

PostgreSQL 元组头的结构在 HeapTupleHeaderData 中记载，此结构体中不记载元组是否被修改行的信息，只用两个标志位“t\_infomask”和“t\_infomask2”<sup>①</sup>来标识本元组加锁的信息，也就是说，单个元组的加锁信息，没有使用单独的内存空间表示，这意味着 PostgreSQL 可以支持任意多个行级锁，这样就不存在很多的元组级的锁把锁表撑死的现象。

PostgreSQL V9.3 之后的版本提供四种行级锁模式，分别为：

- ❑ **更新锁 (FOR UPDATE) 模式：**SELECT 命令中存在“FOR UPDATE”表示将来可能发生更新操作，这使得 SELECT 命令读取的行被锁定。用以防止被锁定的行被其他事务再次锁定、修改或删除，即其他事务尝试在相同元组上执行 UPDATE、DELETE、SELECT FOR UPDATE、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE 或 SELECT FOR KEY SHARE 等操作时，将被阻塞。另外，删除一行、更新一列，也可以获得更新锁模式。当事务隔离级别为 REPEATABLE READ 或者 SERIALIZABLE 时，如果一行被锁定的元组从事务开始起，已经被改变，此事务将抛出一个错误；
- ❑ **无键更新锁 (FOR NO KEY UPDATE) 模式：**这种模式与 FOR UPDATE 相似，但是封锁的粒度更弱，此锁不阻塞 SELECT FOR KEY SHARE 锁模式加在同样的行上。它通过不获取更新锁 (FOR UPDATE) 的 UPDATE 命令获得；
- ❑ **共享锁 (FOR SHARE) 模式：**这种模式与无键更新锁类似，可以在获取的行上加共享锁（不是加排他锁）。此锁阻止其他事务在同样的行上进行 UPDATE、DELETE、SELECT FOR UPDATE 或 SELECT FOR NO KEY UPDATE 操作，但并不阻止它们进行 SELECT FOR SHARE 或 SELECT FOR KEY SHARE 操作；
- ❑ **键共享锁 (FOR KEY SHARE) 模式：**这种模式行为类似于共享锁 (FOR SHARE)，但封锁的粒度弱于共享锁 (FOR SHARE)。此锁可以阻止 SELECT FOR UPDATE，但不阻止 SELECT FOR NO KEY UPDATE。键共享锁阻止其他事务对同样的行进行 DELETE 或任何更改该索引键值的 UPDATE，但不排斥任何其他的 UPDATE、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE 或者 SELECT FOR KEY SHARE 操作。

① 参见9.1.2节。



2. 行级锁之间的相容性

如表 8-10 所示，可以看出，行级锁的四种模式在不同的事务之间，多数是互斥的。

表 8-10 行级锁相容性表

Requested Lock Mode	Current Lock Mode(X 表示新的锁申请被当前锁持锁者排斥)			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE	Y	Y	Y	X
FOR SHARE	Y	Y	X	X
FOR NO KEY UPDATE	Y	X	X	X
FOR UPDATE	X	X	X	X

3. 行级锁的加锁和释放

如图 8-10 所示，行级锁被使用在 DELETE ( heap\_delete() 函数)、UPDATE ( heap\_update() 函数) 操作上，或者使用在显式加锁或触发器相关操作上。

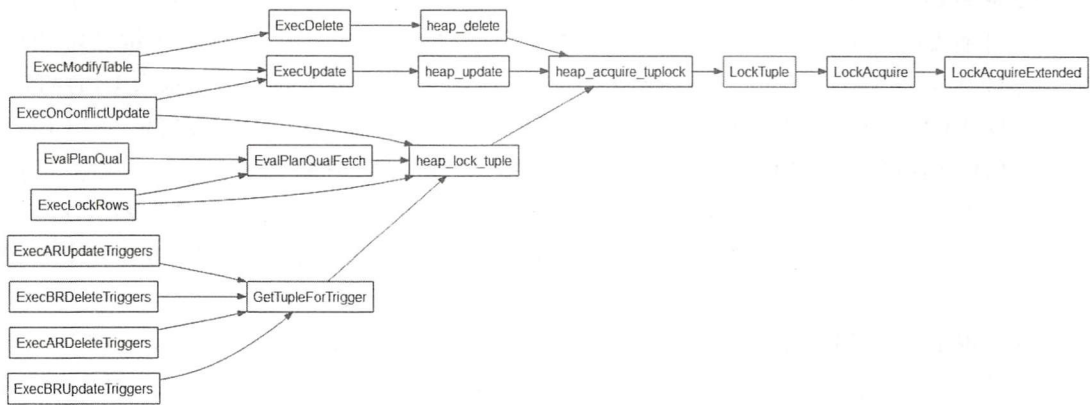


图 8-10 元组加锁操作调用上下文

元组级别的锁，简称元组锁，用于控制对用户数据的并发访问，V9.3 之后的版本提供四种行级锁模式，如前已经表述。

元组锁被数据库的事务管理系统自动施加或被用户用 SQL 语句显式施加，但不管是以任何方式加的锁，在事务提交或回滚阶段自动释放。

元组锁的加锁和释放锁类似表锁、页锁的封锁过程，也是调用 LockAcquire() 函数和 LockRelease() 函数完成。

设置行级锁的标志，是通过宏 SET\_LOCKTAG\_TUPLE 完成的。

```
#define SET_LOCKTAG_TUPLE(locktag,dboid,relid,blocknum,offnum) \
    ((locktag).locktag_field1 = (dboid), \
    (locktag).locktag_field2 = (relid), \
    (locktag).locktag_field3 = (blocknum), \
    (locktag).locktag_field4 = (offnum), \
```

```
(locktag).locktag_type = LOCKTAG_TUPLE, \    //表明是行级锁
(locktag).locktag_lockmethodid = DEFAULT_LOCKMETHOD)
```

行锁的施加动作,是通过 LockTuple() 函数完成,其过程很简单,通过调用上述的宏先设置行锁标志,然后调用 LockAcquire() 函数完成加锁操作。

```
void
LockTuple(Relation relation, ItemPointer tid, LOCKMODE lockmode)
{...
    SET_LOCKTAG_TUPLE(tag, //设置行锁标志
        relation->rd_lockInfo.lockRelId.dbId, relation->rd_lockInfo.lockRelId.relId,
        ItemPointerGetBlockNumber(tid), ItemPointerGetOffsetNumber(tid));

    (void) LockAcquire(&tag, lockmode, false, false); //申请行锁
}
```

行锁的释放动作,是通过 UnLockTuple() 函数完成,其过程也简单,通过调用上述的宏先设置锁标志,然后调用 LockRelease() 函数完成释放锁操作。

尽管加锁操作是一个函数,但是在 PostgreSQL 中被封装为一个宏(解锁操作相似),定义如下:

```
#define LockTupleTuplock(rel, tup, mode) \
    LockTuple((rel), (tup), tupleLockExtraInfo[mode].hwlock)
```

宏 LockTupleTuplock 又被 heap\_acquire\_tuplock() 函数封装,完成加锁操作。下面我们就以 UPDATE 操作来观察元组级锁加锁的过程。

heap\_update() 函数用于对元组加锁,注意封锁的粒度在于元组一级。从如下的代码中可以看出,元组级的锁,是直接施加的,不存在锁之间进行升级和降级的操作。DELETE 操作在元组上加锁的思路类似 UPDATE 操作,不再举例。

```
HTSU_Result
heap_update(Relation relation, ItemPointer otid, HeapTuple newtup, //更新操作
    CommandId cid, Snapshot crosscheck, bool wait, HeapUpdateFailureData
    *hufd, LockTupleMode *lockmode)
{...
    HeapSatisfiesHOTandKeyUpdate(relation, hot_attrs, key_attrs, id_attrs,
    //根据是否存在对索引列的更新操作确定在元组上的加锁模式
        &satisfies_hot, &satisfies_key, //返回值
        &satisfies_id, &oldtup, newtup);

    if (satisfies_key)
    {
        *lockmode = LockTupleNoKeyExclusive; //不更新索引列
    }
    else
    {

```



```

        *lockmode = LockTupleExclusive; //更新索引列
    ...
}
...
if (result == HeapTupleInvisible) //被更新对象不可见, 报错, 事务回滚
{...}
else if (result == HeapTupleBeingUpdated && wait)
//元组正被并发的其他事务更新⊖, 且参数wait指定需要等待
{...
    if (infomask & HEAP_XMAX_IS_MULTI) //有并发事务在操作同一个元组
    {...
        if (DoesMultiXactIdConflict((MultiXactId) xwait, infomask,
//是否存在并发的事务已经施加了同样的锁
                                *lockmode))
        {...
            /* acquire tuple lock, if necessary */
            heap_acquire_tuplock(relation, &(oldtup.t_self), *lockmode,
//锁兼容则可以直接加锁, 注意施加的是元组锁
                                LockWaitBlock, &have_tuple_lock);
//LockWaitBlock表明发出加锁请求, 等待锁被释放
        ...
    }
}
...
}
else if (TransactionIdIsCurrentTransactionId(xwait))
//请求加锁者就是自己, 则不必再加锁
{...}
else if (HEAP_XMAX_IS_KEYSHR_LOCKED(infomask) && key_intact)
//请求加锁者只申请“key-share”而UPDATE不改变索引列, 则不必再加锁
{...}
else //当以上都不是的时候, 正是可以为元组加锁的时机
{...
    heap_acquire_tuplock(relation, &(oldtup.t_self), *lockmode,
//直接加锁, 注意施加的是元组锁
                                LockWaitBlock, &have_tuple_lock);
    XactLockTableWait(xwait, relation, &oldtup.t_self, XLTW_Update);
//等待阻碍本事务的其他事务提交或回滚
...
}
...
}
...
CheckForSerializableConflictIn(relation, &oldtup, buffer);
//SSI机制, 检查是否存在一个“rw-conflict”

```

⊖ 如果元组没有并发的DML类操作, 则更新元组时, 是没有必要加元组锁的。换句话说, 两个并发的更新语句更新同一个对象, 第一个更新不会加元组锁, 而第二个事务则需要通过加元组锁来保证同一个数据对象不会被并发操作破坏数据的一致性。



```
...
    if (have_tuple_lock)
        UnlockTupleTuplock(relation, &(oldtup.t_self), *lockmode);
        //更新完毕，解锁（和SS2PL机制不同之处）。注意这里是DML类操作
...
    return HeapTupleMayBeUpdated;
}
```

这里需要特别注意，行级锁的施加是为了防止并发修改出现数据不一致，而行级锁施加之后，立即释放，这一点与 SS2PL 不同。而执行的操作是 DML 操作，也就是说，DML 类型的行级锁操作不遵守 SS2PL 封锁协议（参见 7.3.2 和 7.3.4 节，注意其中的锁是在事务标识为提交或回滚状态之后才释放的，但是释放的是 DDL 类型的锁，非 DML 类型的锁）。

8.3.4 Advisory locks (劝告锁)

PostgreSQL 提供了一种方式，由用户在自己的应用中、创建具有“自定义”含义的锁，这种锁被称为劝告锁（advisory lock）。“自定义”是指用户指定在什么对象（或称之为资源）上加锁，此锁的含义由用户自己确定。

劝告锁的生命周期，自加锁开始，会一直保持到被显式释放或会话结束。劝告锁与会话层或事务层没有直接的关系，即在会话层级获得的劝告锁，不受事务的 COMMIT 或 ROLLBACK 操作的影响，劝告锁请求并不遵守事务语义。

劝告锁的加锁模式有共享锁、排它锁两种。劝告锁可以被拥有它的进程多次获取；对于每个完成的锁请求，在锁被真正释放前一定要有一个对应的解锁请求，即加锁几次，相应地要释放锁几次。

PostgreSQL 提供的劝告锁函数如表 8-11 所示。

表 8-11 PostgreSQL 劝告锁函数表

函数功能	加锁函数	释放锁函数
在本 session 中为指定的 pid 加锁	pg_blocking_pids()	
加 ExclusiveLock 锁	pg_advisory_lock(key bigint) pg_advisory_lock(key1 int, key2 int)	pg_advisory_unlock(key bigint) pg_advisory_unlock(key1 int, key2 int)
加 ShareLock 锁	pg_advisory_lock_shared(key bigint) pg_advisory_lock_shared(key1 int, key2 int)	pg_advisory_unlock_shared(key bigint) pg_advisory_unlock_shared(key1 int, key2 int)
加 ExclusiveLock 锁	pg_advisory_xact_lock(key bigint) pg_advisory_xact_lock(key1 int, key2 int)	
加 ShareLock 锁	pg_advisory_xact_lock_shared(key bigint) pg_advisory_xact_lock_shared(key1 int, key2 int)	

(续)

函数功能	加锁函数	释放锁函数
尝试加 ExclusiveLock 锁	pg_try_advisory_lock(key bigint) pg_try_advisory_lock(key1 int, key2 int)	带有 try 的函数如果加锁不成功,则不等待,直接返回,所以没有对应的释放锁的函数
尝试加 ShareLock 锁	pg_try_advisory_lock_shared(key bigint) pg_try_advisory_lock_shared(key1 int, key2 int)	
尝试加 ExclusiveLock 锁	pg_try_advisory_xact_lock(key bigint) pg_try_advisory_xact_lock(key1 int, key2 int)	
尝试加 ShareLock 锁	pg_try_advisory_xact_lock_shared(key bigint) pg_try_advisory_xact_lock_shared(key1 int, key2 int)	
释放本 SESSION 施加的所有 advisory 锁		pg_advisory_unlock_all()

另外, 劝告锁在使用“劝告锁”的相关函数之间才发生作用。例如, 劝告锁的一个示例如表 8-12 所示。

表 8-12 劝告锁示例表

SESSION 1	SESSION 2	说明
select * from parent; pid   a1   b1 -----+----- 1   1   1 2   2   2 3   3   3		表 parent 初始有三条元组
BEGIN; SELECT pg_advisory_lock(pid) FROM parent WHERE pid=3; SELECT * FROM active_locks;		
	BEGIN; SELECT pg_advisory_lock(pid) FROM parent;	SESSION 2 的 SELECT 被阻塞, 不能继续执行。如果查询语句为“SELECT pid FROM parent”, 则在列对象 pid 上的查询不被阻塞, 即 SESSION 2 不会被 SESSION 使用的 pg_advisory_lock() 操作所加的锁阻塞
ROLLBACK;		SESSION 2 的 SELECT 继续被阻塞, 不能继续执行
SELECT pg_advisory_unlock_all();		SESSION 1 释放所加的锁, SESSION 2 的 SELECT 不再被阻塞, 得以继续执行

## 8.4 事务锁的管理

锁的基本操作，包括锁的施加、授予、释放，以及冲突检测和死锁检测。死锁检测因在锁管理中占有重要地位，8.5节单独讲述。本节讨论锁的施加、释放等操作的实现过程。

### 8.4.1 获取锁

`LockAcquireExtended()` 函数负责获取锁的流程处理，被上层函数如施加页面锁等包装了加锁逻辑的函数调用，实现真正的加锁操作。

加锁请求需要结果验证是否存在死锁冲突，如果不存在，则可以被授予。

准备施加的锁，先在本地锁表中寻在是否存在，如果不存在，再从进程槽上寻找，如果还是不存在，则从全局的共享缓冲中寻找，找不到则可以创建。详细过程如下面代码分析的六个主要步骤：

```
/*
 * LockAcquireExtended - allows us to specify additional options
 *
 * reportMemoryError specifies whether a lock request that fills the lock table
should generate an ERROR or not.
 * This allows a priority caller to note that the lock table is full and then
begin taking extreme action
 * to reduce the number of other lock holders before retrying the action. */
LockAcquireResult
LockAcquireExtended(const LOCKTAG *locktag, LOCKMODE lockmode,
//这两个参数表示要在哪个对象上施加什么样的锁
                    bool sessionLock, bool dontWait, bool reportMemoryError)
{...//第一步：从本地锁表中找锁
    locallock = (LOCALLOCK *) hash_search(LockMethodLocalHash, (void *)
&localtag, HASH_ENTER, &found);
    //如果没有找到，则是一个新的本地锁对象，初始化此锁；否则，如果本地锁表满，则扩容一倍
    ...
    if (lockmode >= AccessExclusiveLock && locktag->locktag_type == LOCKTAG_RELATION &&
        !RecoveryInProgress() && XLogStandbyInfoActive()) //在关系上施加锁，需要同步
        到 “a standby server”
    {
        LogAccessExclusiveLockPrepare();
        log_lock = true;
    }
    //第二步：用快速方法找锁
    //进程的结构体（参见8.2.2节对于进程结构体的说明）上提供了一些事务锁槽，用以加快事务的锁操
    作。获取事务锁时，优先使用在进程结构体上
    if (EligibleForRelationFastPath(locktag, lockmode) && FastPathLocalUseCount
        < FP_LOCK_SLOTS_PER_BACKEND) //提供的快速访问锁的机制
    {...
```



```

    if (FastPathStrongRelationLocks->count[fasthashcode] != 0)
        acquired = false;
    else
        acquired = FastPathGrantRelationLock(locktag->locktag_field2, lockmode);
        //是否可以快速获得锁
    LWLockRelease(&MyProc->backendLock);
    if (acquired) //如果快速模式可用, 则调用GrantLockLocal()函数赋予锁
    {
        locallock->lock = NULL;
        locallock->proclock = NULL;
        GrantLockLocal(locallock, owner); //locallock是注册在本地锁表里的, 如本函
        数初始之处。本函数的作用就是把准备授予的本地锁的属性修改, 让锁上的计数器和锁的拥有者的计数器都增
        长, 且建立锁属主和锁的关系(解锁和判断死锁时候要使用)
        return LOCKACQUIRE_OK;
    }
}

//第三步: 用快速方法到其他进程上找锁
if (ConflictsWithRelationFastPath(locktag, lockmode))
//想要申请的锁, 其快速访问模式已经被其他进程抢先获得
{...
    BeginStrongLockAcquire(locallock, fasthashcode);
    if (!FastPathTransferRelationLocks(lockMethodTable, locktag, hashcode))
        //如果不能分配(需在全局锁表中分配), 则失败退出
    {...}
}

... //第四步: 在全局锁表上找锁, 找不到则创建
proclock = SetupLockInTable(lockMethodTable, MyProc, locktag, ashcode, lockmode);
if (!proclock){...} //申请失败, 报错退出
locallock->proclock = proclock;
//申请成功, 全局共享缓冲的锁表和进程等信息写入本地锁表的元素中,
lock = proclock->tag.myLock; //相当于同一个锁对象在本地锁表和全局锁表中都存在
locallock->lock = lock;

//第五步: 检查是否存在锁冲突
/* If lock requested conflicts with locks requested by waiters, must join wait queue.
 * Otherwise, check for conflict with already-held locks.
 * (That's last because most complex check.) */
if (lockMethodTable->conflictTab[lockmode] & lock->waitMask)
//申请的锁与锁的等待者申请的锁冲突, 则本次申请不可成功
    status = STATUS_FOUND;
else
    status = LockCheckConflicts(lockMethodTable, lockmode, lock, proclock);
    //检查是否存在冲突, 参见8.4.4节

//第六步: 如果锁不冲突则授予, 如果有冲突则根据本函数的入口参数确定是否等待以及进行死锁检测
if (status == STATUS_OK) //新申请的锁没有冲突, 可以授予

```

```

{
    /* No conflict with held or previously requested locks */
    GrantLock(lock, proclock, lockmode); //新申请的锁的粒度即锁类型注册到锁对象上
    GrantLockLocal(locallock, owner);
    //锁上的计数器和锁的拥有者的计数器都增长, 且建立锁属主和锁的关系 (解锁和判断死锁时候要使用)
}
else //新申请的锁有冲突, 不可以授予
{...
    if (dontWait){...} //申请不成功时指定不等待, 执行一些清理工作后返回到上层函数。
    多数加锁申请都不等待, 只有VACUUM等类操作需要等待
    //否则, 申请不成功时指定等待
    MyProc->heldLocks = proclock->holdMask;
    //Set bitmask of locks this process already holds on this object.
...
    WaitOnLock(locallock, owner); // Sleep till someone wakes me up.
    //等待锁被授予。并进行死锁检测 (参见8.5节)
...
}
...
return LOCKACQUIRE_OK;
}

```

PostgreSQL 官方文档给出如下一段话, 很好地解释了加锁原则。

```

1. A lock request is granted immediately if it does not conflict with
//1 新申请的锁, 与已经授予的和处于等待状态的锁不冲突, 则可授予
any existing or waiting lock request, or if the process already holds an
//2 本进程持有同样对象的锁, 则可授予)
instance of the same lock type (eg, there's no penalty to acquire a read
lock twice). Note that a process never conflicts with itself, eg one
//3 一个事务内的锁不互斥
can obtain read lock when one already holds exclusive lock.

2. Otherwise the process joins the lock's wait queue. Normally it will
//4 上述加锁不成功, 则进入锁等待队列的尾部
be added to the end of the queue, but there is an exception: if the
//5 一个优化点是: 如果本进程已经在同一个对象上持有锁,
process already holds locks on this same lockable object that conflict
// 此对象上已经分配的锁与其他正处于等待的加锁请求冲突, 那么:
with the request of any pending waiter, then the process will be
// 新申请的锁不进入锁等待队列的尾部, 而是插入到正处于等待的、
inserted in the wait queue just ahead of the first such waiter.
(If we // 与已经分配的锁冲突的加锁请求之前。
did not make this check, the deadlock detection code would adjust the
// 这个优化是死锁检测过程中完成的。
queue order to resolve the conflict, but it's relatively cheap to make
the check in ProcSleep and avoid a deadlock timeout delay in this case.)

```



Note special case when inserting before the end of the queue: if the process's request does not conflict with any existing lock nor any waiting request before its insertion point, then go ahead and grant the lock without waiting.

### 8.4.2 锁查找或创建

当一个加锁请求来临的时候，而本地锁表里又不存在时，就可以从全局锁表里找出一个，找不到则创建一个，创建不成功则意味着全局锁表已经满了（全局锁表位于共享内存，大小有限），不能再创建。

```

/*
 * Find or create LOCK and PROCLock objects as needed for a new lock request.
 *
 * Returns the PROCLock object, or NULL if we failed to create the objects for
 * lack of shared memory.
 * The appropriate partition lock must be held at entry, and will be held at exit. */
static PROCLock *
SetupLockInTable(LockMethod lockMethodTable, PGPROC *proc,
                 const LOCKTAG *locktag, uint32 hashcode, LOCKMODE lockmode)
//指定锁请求和hash值，查找是否存在对应的锁
{...
    //Find or create a lock with this tag.      从全局锁表LockMethodLockHash里找
    lock = (LOCK *) hash_search_with_hash_value(LockMethodLockHash,
        (const void *) locktag, hashcode, HASH_ENTER_NULL, &found);
    if (!lock) //全局锁表LockMethodLockHash已满
        return NULL;
    if (!found) //没有找到但是创建成功，需要初始化
    {
        lock->grantMask = 0;
        lock->waitMask = 0;
        SHMQueueInit(&(lock->procLocks));
        ProcQueueInit(&(lock->waitProcs));
    }
    else //找到
    {
        ...
    }
    //锁对象找到以后，则需要建立锁和进程之间的关系
    procllock = (PROCLock *) hash_search_with_hash_value(LockMethodProcLockHash,
        (void *) &procllocktag, procllock_hashcode, HASH_ENTER_NULL, &found);
    if (!procllock) //共享缓冲进程太多，队列满
    {
        if (lock->nRequested == 0) //锁还没有被使用，则从全局锁表中移除，
            因为LockMethodProcLockHash已经满了，得不到锁对应的procllock
    }
}

```



```

    {...
        if (!hash_search_with_hash_value(LockMethodLockHash, (void *) &
            (lock->tag), hashcode, HASH_REMOVE, NULL))
            elog(PANIC, "lock table corrupted");
    }
    return NULL;
}
if (!found) //新建的prolock对象, 初始化这个对象
{
    SHMQueueInsertBefore(&lock->procLocks, &proclock->lockLink);
    SHMQueueInsertBefore(&(proc->myProcLocks[partition]), &proclock->procLink);
    PROCLOCK_PRINT("LockAcquire: new", proclock);
}
else //根据宏的定义情况, 确定是否进行可能的死锁检测, 但不是PostgreSQL默认通过的死锁检测方式
{
    ...
}

lock->nRequested++; //锁申请成功, 则开始计数
lock->requested[lockmode]++;
...
return proclock;
}

```

### 8.4.3 释放锁

LockRelease() 函数释放锁, 是一个物理操作, 被上层带有逻辑语义的释放锁操作调用如 UnlockTuple() 函数调用了 LockRelease() 函数, 完成指定了锁粒度的释放工作。锁释放之后, 需要检查是否可以唤醒等待队列中的会话或系统进程。

```

/* LockRelease -- look up 'locktag' and release one 'lockmode' lock on it.
 *      Release a session lock if 'sessionLock' is true, else release a regular transaction lock.
 *
 * Side Effects: find any waiting processes that are now wakable, grant them their
requested locks and awaken them.
 *      (We have to grant the lock here to avoid a race between the waking
process and any new process to
 *      come along and request the lock.) */
bool
LockRelease(const LOCKTAG *locktag, LOCKMODE lockmode, bool sessionLock)
//在lockmode对象上释放lockmode指定粒度的锁
(...//从本地锁表中找要释放的锁
    locallock = (LOCALLOCK *) hash_search(LockMethodLocalHash, (void *)
        &localtag, HASH_FIND, NULL);
... //没有找到则报错
    { //处理ResourceOwner和Lock的关系
    ...
}

```

```

    }
    locallock->nLocks--; //持锁次数递减, 如果值大于零, 表示这个锁还需要继续保持
                        // (还有其他粒度的锁存在)
    if (locallock->nLocks > 0)
        return TRUE;

    /* Attempt fast release of any lock eligible for the fast path. */
    if (EligibleForRelationFastPath(locktag, lockmode) && FastPathLocalUseCount > 0)
        //存在可快速释放的锁
    {
        LWLockAcquire(&MyProc->backendLock, LW_EXCLUSIVE);
        released = FastPathUnGrantRelationLock(locktag->locktag_field2, lockmode);
        LWLockRelease(&MyProc->backendLock);
        if (released)
        {
            RemoveLocalLock(locallock);
            return TRUE;
        }
    }
    ...
    if (!lock) //从全局锁表中, 找出lock和prolock两种对象
    {
        lock = (LOCK *) hash_search_with_hash_value(LockMethodLockHash,
            (const void *) locktag, locallock->hashcode, HASH_FIND, NULL);
        if (!lock)
            elog(ERROR, "failed to re-find shared lock object");
        ...
        locallock->proclock = (PROCLOCK *) hash_search(LockMethodProcLockHash,
            (void *) &proclocktag, HASH_FIND, NULL);
        if (!locallock->proclock)
            elog(ERROR, "failed to re-find shared proclock object");
    }
    ... //找到之后, 释放锁, 并唤醒等待此锁的会话进程
    wakeupNeeded = UnGrantLock(lock, lockmode, proclock, lockMethodTable); //找到之后,
    //释放锁。并决定是否唤醒其他等待此锁的会话进程 (唤醒的条件是: 等待的队列中有申请
    此处需要释放的锁的类型)
    CleanUpLock(lock, proclock, lockMethodTable, locallock->hashcode, wakeupNeeded);
    //释放锁之后, 唤醒等待此锁的会话进程
    ...
    RemoveLocalLock(locallock); //从本地锁表中去掉此锁请求
    return TRUE;
}

```

## 8.4.4 锁冲突检测

依据锁相容性表和在进程上已经持有的锁的情况进行锁冲突检测, 这是常规的判断方



式。PostgreSQL 特殊之处，在于其提供了并行的进程完成同一个任务，即对于同一个事务，可以由多个不同的进程协作完成，这样的多个进程会被归为一个进程组。因此需要考虑锁在不同进程中是否与同一个进程组中的进程所施加的锁之间的相容关系。

```

/*
 * LockCheckConflicts -- test whether requested lock conflicts with those already granted
 * Returns STATUS_FOUND if conflict, STATUS_OK if no conflict.
 *
 * NOTES:
 *     Here's what makes this complicated: one process's locks don't conflict
 *     with one another,
 *     no matter what purpose they are held for (eg, session and transaction locks do
 *     not conflict).
 *     Nor do the locks of one process in a lock group conflict with those of another
 *     process in the same group.
 *     So, we must subtract off these locks when determining whether the requested
 *     new lock conflicts with those already held. */
int
LockCheckConflicts(LockMethod lockMethodTable, LOCKMODE lockmode, LOCK *lock,
PROCLOCK *proclock)
{
    ...//conflictMask, 是所申请的锁粒度lockmode在相容性锁表lockMethodTable-
    >conflictTab[lockmode]里的掩码
    if (!(conflictMask & lock->grantMask)) //与已经被授予的锁粒度（锁类型）不冲突，
    则可以授予。注意
    {
        PROCLOCK_PRINT("LockCheckConflicts: no conflict", proclock);
        return STATUS_OK;
    }
    //有冲突，则检查是不是本进程或本组并发进程持有冲突的锁，如果是，则是一个事务，允许加锁
    myLocks = proclock->holdMask;
    for (i = 1; i <= numLockModes; i++) //遍历每种类型的锁
    {
        if ((conflictMask & LOCKBIT_ON(i)) == 0) //第i位为0，会与任何锁冲突，申请的锁
        就不会被授予，所以计数器值为0
        {
            conflictsRemaining[i] = 0;
            continue;
        }
        conflictsRemaining[i] = lock->granted[i]; //第i位为1，即存在被授予锁的可能性，
        从已经授予的锁找出授予次数
        if (myLocks & LOCKBIT_ON(i)) //第i位为1，且就是所申请的锁，则没有冲突
            --conflictsRemaining[i];
        totalConflictsRemaining += conflictsRemaining[i]; //汇总所有的冲突次数
    }

    /* If no conflicts remain, we get the lock. */

```



```

if (totalConflictsRemaining == 0) //没有冲突
{
    ...
    return STATUS_OK;
}

/* If no group locking, it's definitely a conflict. */ //冲突总数不是零，且不是并
行的会话进程组，则冲突一定存在
if (proclock->groupLeader == MyProc && MyProc->lockGroupLeader == NULL)
{
    ...
    return STATUS_FOUND;
}

/* Locks held in conflicting modes by members of our own lock group are not
real conflicts;
* we can subtract those out and see if we still have a conflict.
* This is O(N) in the number of processes holding or awaiting locks on this object.
* We could improve that by making the shared memory state more complex (and
larger) but it doesn't seem worth it. */
procLocks = &(lock->procLocks); //遍历申请lock这个锁的所有会话进程组。注意包括了存在
并行的会话进程组的情况
otherproclock = (PROCLOCK *) SHMQueueNext(procLocks, procLocks,
offsetof(PROCLOCK, lockLink));
while (otherproclock != NULL)
{
    if (proclock != otherproclock &&
        proclock->groupLeader == otherproclock->groupLeader &&
        //并行的会话进程组的情况
        (otherproclock->holdMask & conflictMask) != 0)
        //并行的会话进程组持有的锁与冲突掩码相容
    {
        int intersectMask = otherproclock->holdMask & conflictMask;
        //得到与并行的会话进程组持有的锁与冲突掩码相容的锁粒度
        for (i = 1; i <= numLockModes; i++)
        {
            if ((intersectMask & LOCKBIT_ON(i)) != 0) //相容
            {
                if (conflictsRemaining[i] <= 0)
                    elog(PANIC, "proclocks held do not match lock");
                conflictsRemaining[i]--;
                totalConflictsRemaining--;
            }
        }
    }

    if (totalConflictsRemaining == 0) //没有不相容的，锁可以授予
    {
        PROCLOCK_PRINT("LockCheckConflicts: resolved (group)", proclock);
    }
}

```

```

        return STATUS_OK;
    }
}

otherproclock = (PROCLock *) SHMQueueNext(procLocks, &otherproclock->
lockLink, offsetof(PROCLock, lockLink));

/* Nope, it's a real conflict. */
PROCLock_PRINT("LockCheckConflicts: conflicting (group)", proclock);
return STATUS_FOUND; //存在冲突
}

```

## 8.5 死锁检测

PostgreSQL 使用等待图进行死锁检测，但是因支持了并行操作而更为复杂。因为同属于一个事务的多个进程可能会并行发出不同的加锁请求，发自同一个事务的多个加锁请求会和其他的进程存在多个冲突，所以死锁检测更为困难。

### 8.5.1 数据结构

#### 1. 死锁检测的状态

```

/* Deadlock states identified by DeadLockCheck() */
//死锁检测过程中，锁处于的可能的状态如下
typedef enum
{
    DS_NOT_YET_CHECKED, /* no deadlock check has run yet */
                        //还没有进行死锁检测，检测前的一个临时状态
    DS_NO_DEADLOCK,     /* no deadlock detected */
                        //不存在死锁
    DS_SOFT_DEADLOCK,   /* deadlock avoided by queue rearrangement */
                        //通过等待队列的重新排队，死锁被避免。一种优化技术
    DS_HARD_DEADLOCK,   /* deadlock, no way out but ERROR */
                        //存在死锁
    DS_BLOCKED_BY_AUTOVACUUM /* no deadlock; queue blocked by autovacuum worker */
                        //不存在死锁，只是被Autovacuum进程阻塞了
} DeadLockState;

```

#### 2. 等待图中边的分类

在一个进程的锁等待队列中，处于等待状态的多个进程之间，可能也存在锁冲突，这种情况，增加了等待图的复杂性。

PostgreSQL 使用“soft edge”软边和“hard edge”硬边区分了等待图中的两种边。假



设有 A 和 B 两个进程：

- “hard edge” 硬边：B 进程已经持有的锁和 A 进程准备申请的锁冲突，这时，等待图中存在  $A \rightarrow B$  的一条边，这样的边就是“hard edge”。
- “soft edge” 软边：A 和 B 进程都处于同一个等待队列，B 在前 A 在后，而且 A 和 B 进程准备请求的锁之间冲突，这时，等待图中存在  $A \rightarrow B$  的一条边，这样的边就是“soft edge”。ProcLockWakeup() 函数在执行等待进程唤醒操作的时候，不能优先唤醒 A，而是应该唤醒 B。

这两种边，都会产生死锁。因此在死锁检测的状态中，区分了 DS\_SOFT\_DEADLOCK 和 DS\_HARD\_DEADLOCK，分别对应“soft edge”软边和“hard edge”硬边的情况。

对于存在软边的情况，PostgreSQL 通过对等待队列中的进程进行排序，试图找出一个新的加锁顺序（不是按照物理申请锁时进入等待队列的顺序）而新的等待队列中不存在锁冲突，以解决软边构成死锁的情况。这样做可以减少对事务进行回滚操作。具体算法参见 TopoSort() 函数。

## 8.5.2 等待获取锁与死锁处理

新申请的锁，可能不能立刻被授予，需要申请者进行等待，这是获取锁的过程中的重要一环。WaitOnLock() 函数是一层壳，调用 ProcSleep() 函数等待锁。

```
static void
WaitOnLock(LOCALLOCK *locallock, ResourceOwner owner) //等待、以获取锁
{...
    if (ProcSleep(locallock, lockMethodTable) != STATUS_OK)
        //有死锁发生，调用DeadLockReport() 详细汇报死锁信息
        {... }
    ...
}
```

ProcSleep() 函数调用死锁检测函数检测死锁，如果没有死锁发生则让本申请锁的进程处于等待状态。

```
//ProcSleep -- put a process to sleep on the specified lock
int
ProcSleep(LOCALLOCK *locallock, LockMethod lockMethodTable)
{...
    if (myHeldLocks != 0) //本等待进程持有其他的锁
    {...
        proc = (PGPROC *) waitQueue->links.next; //欲申请锁的等待队列
        for (i = 0; i < waitQueue->size; i++) //遍历欲申请锁的等待队列
        {...
            /* Must he wait for me? */
            if (lockMethodTable->conflictTab[proc->waitLockMode] & myHeldLocks)
```



```

//有其他进程在等待我自己 (myHeldLocks)
{
    /* Must I wait for him ? */
    if (lockMethodTable->conflictTab[lockmode] & proc->heldLocks)
        //我自己在等待其他进程持有的某种锁 (proc->heldLocks)
    {
        ...//发现有AB-BA式死锁，记录下来，后面处理
        RememberSimpleDeadLock(MyProc, lockmode, lock, proc);
        early_deadlock = true;
        break;
    }

    /* I must go before this waiter. Check special case. */
    if ((lockMethodTable->conflictTab[lockmode] & aheadRequests) == 0 &&
        //与等待的进程申请的锁不冲突，则可跳出
        LockCheckConflicts(lockMethodTable, lockmode, lock, procllock) == STATUS_OK)
        //注意一个个按序遍历等待本进程的其他进程的申请锁的状态，排在前面的进程如
        //不与新申请的锁冲突，则可把新申请锁的这种状况优先授予锁
    {
        /* Skip the wait and just grant myself the lock. */
        GrantLock(lock, procllock, lockmode);
        GrantAwaitedLock();
        return STATUS_OK;
    }

    /* Break out of loop to put myself before him */
    break; //注意：这里的前提是“有其他进程在等待我自己 (myHeldLocks)”
}

...
}

else //本等待进程不持任何其他的锁
{
    /* I hold no locks, so I can't push in front of anyone. */
    proc = (PGPROC *) &(waitQueue->links);
}

//Insert self into queue, ahead of the given proc (or at tail of queue).
SHMQueueInsertBefore(&(proc->links), &(MyProc->links));
waitQueue->size++;

lock->waitMask |= LOCKBIT_ON(lockmode); //不能获得申请的锁，则只能处于等待

...
MyProc->waitStatus = STATUS_WAITING; //设置处于等待状态
if (early_deadlock) //处理之前发现的死锁
{
    RemoveFromWaitQueue(MyProc, hashcode);
    return STATUS_ERROR;
}
}

```



```

...
    lockAwaited = locallock; // “lockAwaited” 是处于等待其他锁状态的一个重要标志
...
do
{
    if (InHotStandby)
    {
        /* Set a timer and wait for that or for the Lock to be granted */
        ResolveRecoveryConflictWithLock(locallock->tag.lock);
    }
    else
    {
        if (got_deadlock_timeout)
        {
            CheckDeadLock(); //进行死锁检测
            got_deadlock_timeout = false;
        }
        CHECK_FOR_INTERRUPTS();
    }
    myWaitStatus = *((volatile int *) &MyProc->waitStatus);
    //在每轮循环中, 保存等待状态(可能异步地被其他进程唤醒)
    if (deadlock_state == DS_BLOCKED_BY_AUTOVACUUM && allow_autovacuum_cancel)
    //没有发生死锁, 只是被AUTOVACUUM阻塞
    {
        if ((autovac_pgxact->vacuumFlags & PROC_IS_AUTOVACUUM) &&
            !(autovac_pgxact->vacuumFlags & PROC_VACUUM_FOR_WRAPAROUND))
        {
            int pid = autovac->pid; //获取AUTOVACUUM的进程ID
            ...
            if (kill(pid, SIGINT) < 0) //发信号给AUTOVACUUM的进程, 让其停止
            {
                ...
            }
        }
        else
        {
            ...
        }
    }
    if (log_lock_waits && deadlock_state != DS_NOT_YET_CHECKED)
    //否则, 处理各种死锁的情况, 方式是报告错误
    {
        if (deadlock_state == DS_SOFT_DEADLOCK)
            ereport(LOG, ...);
        else if (deadlock_state == DS_HARD_DEADLOCK)
        {
            ereport(LOG, ...);
        }
        ...
    }
} while (myWaitStatus == STATUS_WAITING);

```



```

//只要处于等待状态,就循环,进行死锁检测等如上检查
...
if (MyProc->waitStatus == STATUS_OK)           //获取了锁
    GrantAwaitedLock();
...
}

```

### 8.5.3 死锁检测

死锁检测,就是从等待图中寻找是否构成了环,如果构成环则有死锁发生;否则不存在死锁。判断是否有环存在,是通过 FindLockCycle() 函数完成的。相关的函数上下文,如图 8-10 所示。

```

/*
 * DeadLockCheck -- Checks for deadlocks for a given process
 *
 * This code looks for deadlocks involving the given process. If any are found,
 * it tries to rearrange lock wait queues to resolve the
 * deadlock. If resolution is impossible, return DS_HARD_DEADLOCK --- the caller
 * is then expected to abort the given proc's transaction.
 */
DeadLockState
DeadLockCheck(PGPROC *proc) //检查指定会话进程proc中是否存在死锁
{...
    /* Search for deadlocks and possible fixes */
    if (DeadLockCheckRecurse(proc)) //递归地进行死锁检测,如果存在soft edge,
        则可以消除其对应的死锁情况
    {...
        if (!FindLockCycle(proc, possibleConstraints, &nSoftEdges))
            //发生死锁,但又不存在环,数据库引擎的内存结构可能被异常破坏了
            elog(FATAL, "deadlock seems to have disappeared");
        //只好报告FATAL错误,强制数据库引擎系统退出
        return DS_HARD_DEADLOCK; /* cannot find a non-deadlocked state */
    }
    //接下来,肯定是存在soft edge的情况
    for (i = 0; i < nWaitOrders; i++) /* Apply any needed rearrangements of wait queues */
        //遍历修复后等待队列
        {
            LOCK      *lock = waitOrders[i].lock;
            //被DeadLockCheckRecurse(proc)修复后的等待队列的顺序
            PGPROC      **procs = waitOrders[i].procs;
            int          nProcs = waitOrders[i].nProcs;
            PROC_QUEUE *waitQueue = &(lock->waitProcs);
            ...

            /* Reset the queue and re-add procs in the desired order */

```



```

ProcQueueInit(waitQueue);
for (j = 0; j < nProcs; j++) //重新排定等待队列
{
    SHMQueueInsertBefore(&(waitQueue->links), &(procs[j]->links));
    waitQueue->size++;
}

...

/* See if any waiters for the lock can be woken up now */
ProcLockWakeup(GetLocksMethodTable(lock), lock);
//给予等待队列中的处于等待状态的进程运行的机会
}

/* Return code tells caller if we had to escape a deadlock or not */
if (nWaitOrders > 0)
    return DS_SOFT_DEADLOCK;
else if (blocking_autovacuum_proc != NULL)
    return DS_BLOCKED_BY_AUTOVACUUM;
else
    return DS_NO_DEADLOCK;
}

```

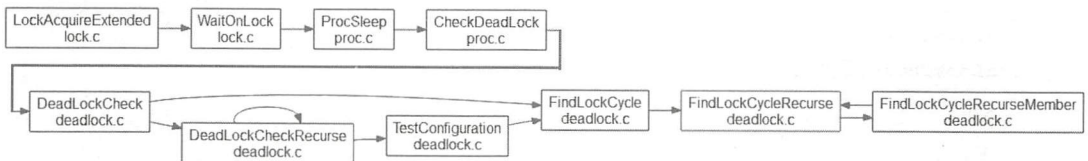


图 8-11 死锁检测函数上下文图

如图 8-11 所示，DeadLockCheckRecurse() 通过调用 TestConfiguration() 再调用 ExpandConstraints() 和 FindLockCycle()，完成的等待图上环的查找，从而确认是否存在死锁。相关代码请参看相关函数，细节不再赘述。

### 8.5.4 进程唤醒

当一个死锁检测完成后，可能存在死锁的 soft edge 被修复为无死锁的情况，不存在死锁的进程就有机会被唤醒以获取锁。除此之外，处于等待队列的进程要想被唤醒，只有在一个事务结束的时候，才能获得被唤醒拿到所申请的锁的机会。相关的调用上下文，如图 8-12 所示。

```

void
ProcLockWakeup(LockMethod lockMethodTable, LOCK *lock)
//唤醒lock对象上的等待队列中的进程
{
    PROC_QUEUE *waitQueue = &(lock->waitProcs); //获得lock对象上的等待队列
}

```

```

...
proc = (PGPROC *) waitQueue->links.next;    //获得lock对象上的等待队列中的进程
while (queue_size-- > 0)                    //遍历lock对象上的等待队列中的进程
{
    LOCKMODE    lockmode = proc->waitLockMode;

    /*
     * Waken if (a) doesn't conflict with requests of earlier waiters, and
     * (b) doesn't conflict with already-held locks.
     */
    if ((lockMethodTable->conflictTab[lockmode] & aheadRequests) == 0 &&
        //与本进程上等待队列中前面的等待者持有的锁不冲突
        LockCheckConflicts(lockMethodTable, //与本进程上已经持有的锁不冲突
                            lockmode, lock, proc->waitProcLock) == STATUS_OK)
    {
        /* OK to waken */
        GrantLock(lock, proc->waitProcLock, lockmode);
        proc = ProcWakeup(proc, STATUS_OK);
    }
    else
    {
        aheadRequests |= LOCKBIT_ON(lockmode); //记录本进程上等待队列中前面的等待
                                                //者持有的锁，供下个等待者下次判断使用
        proc = (PGPROC *) proc->links.next;
    }
}
...
}

```

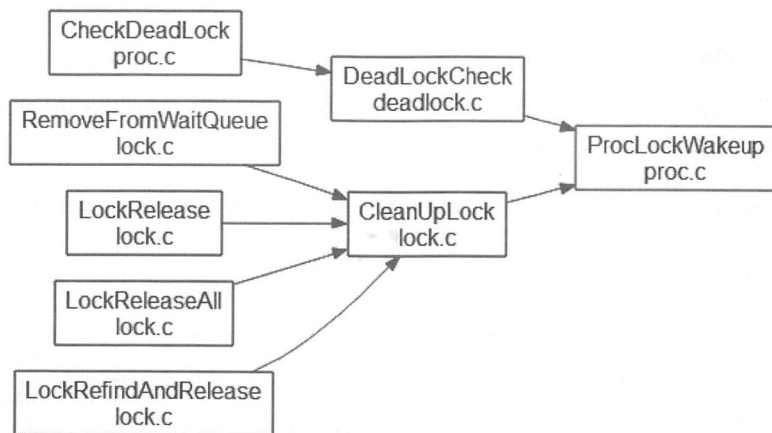


图 8-12 唤醒进程调用关系图



## 8.6 从锁的角度看用法

本节从锁的角度讨论 SQL 语义决定的加锁粒度和加锁内容。

### 8.6.1 AccessShareLock

读取一个对象，需要施加读锁，或称为共享锁。而读锁施加的对象，可以是用户数据，也可以是元数据。元数据的存储形式，是系统表中的元组。所以在代码中可以看到，读锁可以施加在系统表上，也可以施加在用户表上。

#### 1. 为系统表加锁

“AccessShareLock”主要用于为读系统表加锁，此共享锁加在位于共享内存中的系统表上。这类操作很多（同类的函数如 lookup\_ts\_config\_cache() 等），由 heap\_open() 调用 relation\_open() 进一步调用 LockRelationOid() 进一步调用 SetLocktagRelationOid() 和 LockAcquire() 完成在系统表上加 Latch 式的共享锁。LockRelationOid() 函数上下文的调用关系参见图 8-13。

```
find_language_template(const char *languageName) //查找对于指定的编程语言如Perl是否有相
应的语言模板存在
{...
    rel = heap_open(PLTemplateRelationId, AccessShareLock);
    // PLTemplateRelationId值为1136对应的系统表，存放相应的语言模板信息
    ScanKeyInit(&key, Anum_pg_pltemplate_tmplname,
                BTEqualStrategyNumber, F_NAMEEQ,
                NameGetDatum(languageName));
    scan = systable_beginscan(rel, PLTemplateNameIndexId, true, NULL, 1, &key);
    ...
    heap_close(rel, AccessShareLock);
    ...}
```

#### 2. 为用户表加锁

用户表如果只读不写的话（如执行 SELECT 操作），可以加“AccessShareLock”。比如，在用户表上执行 COPY 命令导出数据，执行的函数如下：

```
DoCopy(const CopyStmt *stmt, const char *queryString, uint64 *processed)
{...
    if (stmt->relation)
    {...
        /* Open and lock the relation, using the appropriate lock type. */
        rel = heap_openrv(stmt->relation,
                           (is_from ? RowExclusiveLock : AccessShareLock));
        //如果是从表中COPY数据到数据库外面，则使用共享锁
    ...}
    ...}
```



```

if (rel != NULL)
    heap_close(rel, (is_from ? NoLock : AccessShareLock));
...}

```

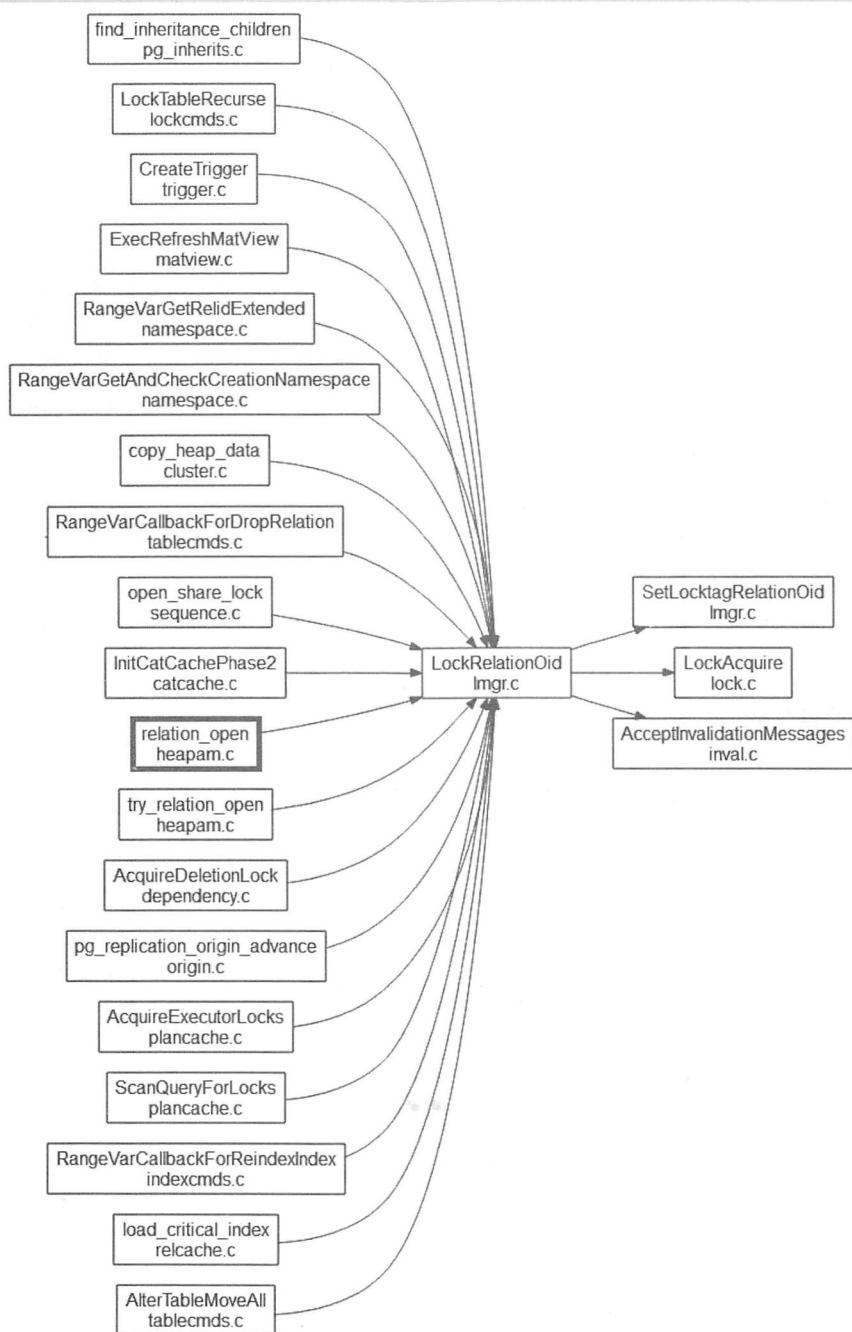


图 8-13 LockRelationOid() 函数调用关系图

## 8.6.2 RowShareLock

“RowShareLock”主要是用于为更新操作而提前锁定目标对象，这里的目标对象就是用户表的数据。但是，“RowShareLock”属于用户表层面的锁而非系统表层面的锁。例如，使用UPDATE语句可以更新的用户表中的数据，提前用“RowShareLock”为执行UPDATE语句而锁定用户表中的数据是允许的，但UPDATE语句不能修改系统表中的元组。而系统表中的元组，则是通过ALTER TABLE等类似的DDL语句完成的。

“RowShareLock”可用于为用户表加共享的读锁，如下所示：

```
RI_FKey_check(TriggerData *trigdata)
//修改外键表上的主外键约束，需要在主键表上进行检查以确定没有违反主外键约束
{...
    /* the relation descriptors of the FK and PK tables.
     *
     * pk_rel is opened in RowShareLock mode since that's what our eventual
     * SELECT FOR KEY SHARE will get on it.
     */
    fk_rel = trigdata->tg_relation;
    pk_rel = heap_open(riinfo->pk_relid, RowShareLock); //在主键表上加共享锁
...}
```

“RowShareLock”也可用于FROM子句中的表对象的读取操作加锁，如ScanQueryForLocks()函数如下：

```
ScanQueryForLocks(Query *parsetree, bool acquire)
{...
    rt_index = 0;
    foreach(lc, parsetree->rtable) //遍历FROM子句中的每一个范围表
    {...
        switch (rte->rtekind)
        {
            case RTE_RELATION: //如果是用户表
                /* Acquire or release the appropriate type of lock */
                if (rt_index == parsetree->resultRelation)
                    lockmode = RowExclusiveLock;
                else if (get_parse_rowmark(parsetree, rt_index) != NULL)
                    lockmode = RowShareLock; //SELECT...FOR UPDATE
                else
                    lockmode = AccessShareLock; //SELECT...FOR SHARE
                if (acquire)
                    LockRelationOid(rte->relid, lockmode);
                else
                    UnlockRelationOid(rte->relid, lockmode);
                break;
            case RTE_SUBQUERY: //如果是子查询，递归遍历
```

```

        /* Recurse into subquery-in-FROM */
        ScanQueryForLocks(rte->subquery, acquire);
        break;
    default:
        /* ignore other types of RTEs */
        break;
    }
}
...}

```

### 8.6.3 RowExclusiveLock

“RowExclusiveLock”用于执行更新、插入、删除类型的操作，但不是所谓的DML类型的操作，因为DML类型的操作对象是用户表中的数据，而对系统表的数据，也可以执行更新、插入、删除操作。

#### 1. 为系统表加锁

“RowExclusiveLock”主要用于为写系统表加锁，排它锁加在位于共享内存中的系统表上。这类操作很多，函数调用关系类似“AccessShareLock”，只是加锁时的模式不同，如为表授权的时候，执行的函数ExecGrant\_Relation如下。

```

ExecGrant_Relation(InternalGrant *istmt) //需要修改系统表，故在内存中直接对系统表加排它锁
{...
    relation = heap_open(RelationRelationId, RowExclusiveLock);
    // RelationRelationId, 值为1259, 标识系统表pg_class
    attRelation = heap_open(AttributeRelationId, RowExclusiveLock);
    // AttributeRelationId, 值为1249, 标识系统表pg_attribute
    ...
    heap_close(attRelation, RowExclusiveLock);
    heap_close(relation, RowExclusiveLock);
    ...}

```

#### 2. 为用户表加锁

```

DoCopy(const CopyStmt *stmt, const char *queryString, uint64 *processed)
{...
    if (stmt->relation)
    {...
        /* Open and lock the relation, using the appropriate lock type. */
        rel = heap_openrv(stmt->relation,
                          (is_from ? RowExclusiveLock : AccessShareLock));
        //如果是COPY数据进入表，则使用排它锁
        ...}
    ...
}

```



```

    if (rel != NULL)
        heap_close(rel, (is_from ? NoLock : AccessShareLock));
    ...}

```

## 8.6.4 ExclusiveLock

“ExclusiveLock”锁抑制并发的粒度很大，仅次于DDL类型的操作。如在各种索引上如GIN、Brin、Gist等上进行操作，再如Btree上执行VACUUM等操作，都需要对被加锁的对象排它访问。

### 1. 创建劝告锁

```

pg_advisory_lock_int8(PG_FUNCTION_ARGS) //在指定对象上加排它式的劝告锁
{...
    SET_LOCKTAG_INT64(tag, key);
    (void) LockAcquire(&tag, ExclusiveLock, true, false); //第四个参数值为true,
    表示劝告锁是Session级的锁，由用户通过函数直接施加。第四个参数值为false，表示获取不到锁
    就不等待
    ...}

```

### 2. 对于诸如VACUUM类似的操作在指定对象上加排它锁

```

btvacuumscan(...) //Btree树(B+树索引)上进行VACUUM类操作
{...
    for (;;)
    {
        /* Get the current relation length */
        if (needLock)
            LockRelationForExtension(rel, ExclusiveLock);
        num_pages = RelationGetNumberOfBlocks(rel);
        if (needLock)
            UnlockRelationForExtension(rel, ExclusiveLock);
    }
    ...
}

```

### 3. 为系统表加锁

```

replorigin_create(char *roname)
{...
    rel = heap_open(ReplicationOriginRelationId, ExclusiveLock);
    // ReplicationOriginRelationId, pg_replication_origin系统表的标识
    ...}

```

#### 4. 为对象加锁

```

GetLockStatusData(void)
{...
    for (i = 0; i < ProcGlobal->allProcCount; ++i)
    {...
        for (f = 0; f < FP_LOCK_SLOTS_PER_BACKEND; ++f)
        {
            LockInstanceData *instance;

            ...

            instance = &data->locks[el];
            SET_LOCKTAG_RELATION(instance->locktag, proc->databaseId, proc->fpRelId[f]);
            instance->holdMask = LOCKBIT_ON(ExclusiveLock);

            ...
        }
    }
}

```

其中, LockInstanceData 定义如下:

```

typedef struct LockInstanceData
{
    LOCKTAG    locktag;        /* 被加锁的对象的标识 */
    LOCKMASK    holdMask;      /* locks held by this PGPROC, 值为
    "LOCKBIT_ON(ExclusiveLock)" 表示被加锁 */
    LOCKMODE    waitLockMode; /* lock awaited by this PGPROC, if any */
    BackendId    backend;      /* backend ID of this PGPROC */
    LocalTransactionId lxid; /* local transaction ID of this PGPROC */
    int          pid;          /* pid of this PGPROC */
    int          leaderPid;     /* pid of group leader; = pid if no group */
    bool         fastpath;      /* taken via fastpath? */
} LockInstanceData;

```

#### 5. 为指定的 Relation 加锁 (物理页面在缓存区的页)

```

RelationGetBufferForTuple(Relation relation, ...) //返回被pin住的且被排它锁锁住的buf页
{...
    if (needLock)
    {
        if (!use_fsm)
            LockRelationForExtension(relation, ExclusiveLock);
        else if (!ConditionalLockRelationForExtension(relation, ExclusiveLock))
        {
            /* Couldn't get the lock immediately; wait for it. */
            LockRelationForExtension(relation, ExclusiveLock); //在relation上加排它锁

```



```

...
        //If some other waiter has already extended the relation, we don't
        need to do so; just use the existing freespace.
        if (targetBlock != InvalidBlockNumber)
        {
            UnlockRelationForExtension(relation, ExclusiveLock);
            goto loop;
        }
    }
}
...
}

```

## 6. 为索引页加锁

```

ginInsertCleanup(GinState *ginst, ...) //为指定的GIN索引页加锁，本质上类似于为指定的Relation加锁
{
    Relation index = ginst->index; //一个index页，也是一个被“Relation”定义的对象
    if (inVacuum)
    {
        //We are called from [auto]vacuum/analyze or gin_clean_pending_list() and
        we would like to wait concurrent cleanup to finish.
        LockPage(index, GIN_METAPAGE_BLKNO, ExclusiveLock); //为指定的索引页加排它锁
    }
    else
    {
        //We are called from regular insert and if we see concurrent cleanup just
        exit in hope that concurrent process will clean up pending list.
        if (!ConditionalLockPage(index, GIN_METAPAGE_BLKNO, ExclusiveLock))
            return;
    }
}

```

## 8.6.5 其他的锁

### 1. ShareUpdateExclusiveLock 锁

“ShareUpdateExclusiveLock”可用于为系统表 pg\_class 等加排它锁，禁止在同一个表对象上同时执行多个类似 ANALYZE 的操作，如：

```

AcquireDeletionLock(const ObjectAddress *object, int flags) //在删除的对象上加排它类的锁
{

```



```

if (object->classId == RelationRelationId) //如果是pg_class系统表
{
    /*
     * In DROP INDEX CONCURRENTLY, take only ShareUpdateExclusiveLock on
     * the index for the moment. index_drop() will promote the lock once
     * it's safe to do so. In all other cases we need full exclusive
     * lock.
     */
    if (flags & PERFORM_DELETION_CONCURRENTLY)
    //如果是DROP INDEX CONCURRENTLY操作，在索引上加可并发访问的排它锁
        LockRelationOid(object->objectId, ShareUpdateExclusiveLock);
    //有部分新请求的锁可以在此锁基础上升级
    else
        LockRelationOid(object->objectId, AccessExclusiveLock);
    //最强级别的锁，没有新的锁可以再升级（即任何锁都被排斥）
}
else
{
    /* assume we should lock the whole object not a sub-object */
    LockDatabaseObject(object->classId, object->objectId, 0, AccessExclusiveLock);
}
}

```

## 2. ShareLock

“ShareLock”锁主要在创建索引时使用，但也有特殊用法，如：

```

XactLockTableWait(TransactionId xid, Relation rel, ItemPointer ctid,
//等待指定的事务提交/回滚操作完成
                    XLTW_Oper oper)
{...
    for (;;)
    {...
        SET_LOCKTAG_TRANSACTION(tag, xid); //xid, 指定的事务
        (void) LockAcquire(&tag, ShareLock, false, false);
        //为了等待指定的事务提交/回滚操作完成，不断加锁（下行解锁）空转
        LockRelease(&tag, ShareLock, false);
    }
}

```

## 3. ShareRowExclusiveLock

“ShareRowExclusiveLock”锁主要用于操作触发器、约束、外键约束等对象。

## 4. AccessExclusiveLock

“AccessExclusiveLock”锁主要用于 DDL 类的操作，此类锁的粒度很大，严禁其他操作并发。

## 8.7 本章小结

PostgreSQL V9.1 版本之前,使用 ReguarLock 完成对“事务中的用户数据”封锁操作,因此,ReguarLock 是 V9.1 之前的唯一类型的事务级的锁。而 SpinLock 和 LWLock 是保障数据库系统的“系统资源”在并发状态下不被破坏,属于系统级的锁。事务级的锁和系统级的锁因保护的对象不同,他们的实现不同,后者简单,前者因带有业务处理的语义 (ACID) 而非常复杂。但事务级的锁是以系统级的锁为基础实现的。

在 PostgreSQL V9.1 版本之前,PostgreSQL 使用 MVCC 机制保障“事务中的用户数据”,读数据的阶段不加锁、只在写数据阶段加锁,这一点,区别于 2PL 技术 (读阶段加锁)。

“事务中的用户数据”,包括两部分内容,一是用户的元数据,存储于系统表中 (如使用 ALTER TABLE...、DROP INDEX 等语句);二是用户的数据,即我们通常所说的用户表里的记录 (如使用 UPDATE、DELETE、VACUUM FULL 语句)。不管是元数据还是用户的数据,一旦其被修改,就需要防止并发操作带来的数据不一致的问题。所以,在数据库内核实现的过程中,通过在元数据所在的系统表上加读锁和排他类型的锁,是必要的;而在用户数据上加锁,我们的思维受到二维表这个表格形式的影响,会直观地认为每一个记录 (即元组) 是一条完整数据,因此并发控制的对象就应该是元组才对。

其实不然,在数据库系统的实现中,每一个记录 (即元组) 如果不是太大,则存放在一个物理的页面当中;太大则分散在不同的物理页面中。IO 操作的基本单位是物理页面,因而数据库为了提高 IO 效率,把数据操作的对象定义为了物理页面,所以数据库的许多操作如事务管理、预先日志管理都是以物理页面为单位进行组织管理的。这样,并发控制模块所操作的对象就由“想当然的元组”转变为“实际上的物理页面”,所以数据库中封锁管理的机制常规的封锁单位,都是物理页面;但是,当一个页面上有多个元组时,在页面上加锁意味许多可能不相关的元组也被加锁,导致并发效率降低,所以数据库中还提供元组级的封锁机制即通常所说的行锁。

在 PostgreSQL V9.1 版本及之后,PostgreSQL 提供了 Serializable Snapshot Isolation (SSI) 机制,用于解决 SI 技术带来的写偏序问题,使得 PostgreSQL 在事务管理机制方面真正具备了可串行化 Serializable 的能力。

总的来说,PostgreSQL 在 MVCC 机制中提供了行锁和页锁,在 SSI 机制中提供了谓词锁,这些锁的使用,保证了 PostgreSQL 正确地实现了事务管理,保证了不同隔离级别下不会出现相应的事务异常。后续的章节,将对事务级的锁的封锁机制进行详细的讲述。

## PostgreSQL并发控制系统的实现——MVCC

PostgreSQL并发控制系统的核心，在于MVCC技术。与MySQL不同的是，PostgreSQL依靠MVCC技术构建了并发访问控制，封锁只是PostgreSQL为实现写并发控制的一个子模块，利用MVCC的多版本和快照以及不占有主导作用的写锁机制，PostgreSQL实现了不完整的并发访问控制机制。

不完整是指PostgreSQL早期只支持快照隔离，而不支持可串行化快照隔离，这样只能支持两种隔离级别（已提交读和可重复读），只有在SSI技术被引进后，才真正支持了可串行化隔离级别。

MySQL是把MVCC和S2PL融合在了一起，使用MVCC技术实现了已提交读和可重复读这两个隔离级别，使用SS2PL技术实现可串行化（参见第10～12章）。而PostgreSQL则完全依赖MVCC（包括SI和行级锁）、SSI来实现事务的并发访问控制。封锁技术对于PostgreSQL的并发访问控制系统而言，只是快照技术的辅助部分。

另外，PostgreSQL历史上的前身POSTGRES，POSTGRES基于SS2PL，只是MVCC的优点<sup>①</sup>巨大，1999年的时候PostgreSQL采用了MVCC。

可串行化是保证数据一致性的必备因素，可串行化在保证可串行化特性时着力于提高并发度。而数据库引擎中传统的可串行化实现技术的方式，有两种：

- ❑ 一是采用S2PL（参见2.2.1节），通过读加锁和严格控制锁的释放，直到事务提交或回滚才释放锁。这样抑制了其他事务对相同数据项进行操作的并发，从而达到可串行化的目的。

---

① 读不阻塞写，写不阻塞读。这个优点，不仅是对于OLAP系统特别有用，对于一个传统的OLTP型系统的意义也非常重大，因为读操作发生不影响写操作的并发，数据可以不断插入、更新。



- ❑ 二是为了避免幻象异常，锁定范围变大。
  - 如 Informix 和 Berkeley DB，通常采用封锁范围大的页锁甚至表锁，避免因谓词范围而锁定较少的元组导致幻象发生，这种方式是通过加大了封锁的范围来避免幻象，可是封锁范围大而抑制了并发度。
  - 再如 MySQL，在索引项上利用 GAP 锁（参见 11.3.1 节“2 锁的种类”），来避免幻象，这种方式，也是通过加大了封锁的范围来避免幻象的，但是与页锁等大粒度的封锁方式相比，单位还是小了很多。GAP 是锁定记录项前的间隙，页锁则包括被锁定的元组和元组之前的间隙，甚至还包括元组之后的间隙和不应锁定的其他元组。

但是，PostgreSQL 基于快照技术的序列化，即 SSI 技术却不同于上述两种方式，其本质，在 9.4 节详述。概括地讲，SSI 技术的本质是基于多版本和行级锁的读写冲突检测机制。相比上述技术，SSI 不需要使用锁来保证可串行化，而是采用运行过程中不断检测读写依赖的方式来实现可串行化的，这样整个数据库引擎的并发度会很高。

所以，本章内容是 PostgreSQL 并发访问控制技术的核心和重点，需要特别注意掌握。

## 9.1 快照

快照，是一个操作可以读写数据的范围。快照保存了当前活动事务，用以影响本事务的读写操作的范围。

### 9.1.1 相关文件

PostgreSQL 实现 MVCC 技术的主要代码，位于表 9-1 所列的文件中，表 9-2 列出了一些主要的函数。

表 9-1 MVCC 技术相关实现文件表	
文件名	功能
snapmgr.c snapmgr.h	快照管理的实现
tqual.c	POSTGRES "time qualification" code, ie, tuple visibility rules
	MVCC 机制下判断元组的可见性
htup_details.h	元组头的定义
snapbuild.c	Infrastructure for building historic catalog snapshots based on contents of the WAL, for the purpose of decoding heapam.c style values in the WAL. 用于逻辑复制中，在这个模块内的其他文件不再列举
predicate.c	SSI 技术的主要实现
snapshot.h	快照的数据结构定义

表 9-2 MVCC 技术实现元组可见性判断的主要函数表

函数名	功能
HeapTupleSatisfiesMVCC()	visible to supplied snapshot, excludes current command
HeapTupleSatisfiesUpdate()	visible to instant snapshot, with user-supplied command counter and more complex result
HeapTupleSatisfiesSelf()	visible to instant snapshot and current command
HeapTupleSatisfiesDirty()	like HeapTupleSatisfiesSelf(), but includes open transactions
HeapTupleSatisfiesVacuum()	visible to any running transaction, used by VACUUM
HeapTupleSatisfiesToast()	visible unless part of interrupted vacuum, used for TOAST
HeapTupleSatisfiesAny()	all tuples are visible

## 9.1.2 数据结构

### 1. 快照的结构

PostgreSQL 使用 `SnapshotData` 表示快照，这样的快照结构体，可以表示多种快照类型。一个快照就是时间线上的多个正处于活动状态（尚未提交或回滚）的事务列表。

```
typedef struct SnapshotData //Struct representing all kind of possible snapshots
{
    // “satisfies” 不同的快照有不同的类型，可能的值为: HeapTupleSatisfiesMVCC /
    HeapTupleSatisfiesHistoricMVCC
    SnapshotSatisfiesFunc satisfies; /* tuple test function */
```

//在事务发生这根线上（时间轴），不断有事务产生、提交或回滚，这表明每一个事务在事务发生这根线上都是一段，这样的段有多个，多个间有重叠有分离，但是，从一个时间段上看，可能看到零个事务、一个事务或多个事务，这样的时间段就是一个快照。所以说，快照是事务发生线条上的一个线段，所以有比这个线段起始点早发生的（xmin之前即小于xmin），也有比这个线段结束点之后发生的（xmax之后即大于xmax）

```
TransactionId xmin; /* all XID < xmin are visible to me */
```

//事务的ID小于快照的左边界xmin，则其对应的数据被快照持有者的事务可见

```
TransactionId xmax; /* all XID >= xmax are invisible to me */
```

//事务的ID大于快照的右边界xmax，则其对应的数据不被快照持有者的事务可见

TransactionId \*xip; //对于“normal MVCC”类型的快照，表示处于活动状态的事务的指针数组；对于“historic MVCC”，表示已经提交的事务的指针数组

```
uint32 xcnt; /* # of xact ids in xip[] */ //xip的计数器
```

```
/*
```

```
* For non-historic MVCC snapshots, this contains subxact IDs that are in
* progress (and other transactions that are in progress if taken during
* recovery). For historic snapshot it contains *all* xids assigned to the
* replayed transaction, including the toplevel xid.
*
```

```
* note: all ids in subxip[] are >= xmin, but we don't bother filtering out
any that are >= xmax
*/
```

```
TransactionId *subxip; //子事务相关的事务ID信息
```



```

    int32          subxcnt;          /* # of xact ids in subxip[] */ //子事务相关的事
务ID信息的计数器
    bool          suboverflowed; /* has the subxip array overflowed? */

    bool          takenDuringRecovery; /* recovery-shaped snapshot? */
    bool          copied;             /* false if it's a static snapshot */
    ...
    /*
     * Book-keeping information, used by the snapshot manager
     */
    uint32        active_count;      /* refcount on ActiveSnapshot stack */
    uint32        regd_count;        /* refcount on RegisteredSnapshots */
    pairingheap_node ph_node;        /* link in the RegisteredSnapshots heap */

    int64          whenTaken;        /* timestamp when snapshot was taken */
    //生成快照时的时间戳值
    XLogRecPtr     lsn;              /* position in the WAL stream when taken */
    //生成快照时的LSN
} SnapshotData;

```

有了这个结构体之后，一个快照对象“Snapshot”就可以定义为一个指针。

```
typedef struct SnapshotData *Snapshot; //Snapshot是一个指针
```

## 2. 元组的结构

PostgreSQL 在一条元组初始部分，定义了如下的系统列，用以表示元组的生命周期。

```

typedef struct HeapTupleFields
{
    //表示一个事务的生命段[t_xmin, t_xmax], t_xmin生于何时, t_xmax卒于何时
    TransactionId t_xmin; /* inserting xact ID */
    //本条元组是由哪个事务创建的（创建者一定时INSERT/COPY...FROM类操作）
    TransactionId t_xmax; /* deleting or locking xact ID */
    //本条元组是由哪个事务删除的

    union
    {
        CommandId      t_cid;        /* inserting or deleting command ID, or both */
        TransactionId t_xvac;        /* old-style VACUUM FULL xact ID */
    } t_field3;
} HeapTupleFields;

```

这几个系统列，被定义在一个完整的元组头结构里。

```

struct HeapTupleHeaderData //元组头
{
    union

```



```

{
    HeapTupleFields t_heap; //t_xmin和t_xmax等系统列
    DatumTupleFields t_datum;
} t_choice;

ItemPointerData t_ctid; /* current TID of this or newer tuple (or a *
speculative insertion token) */ //记录当前元组或者新元组的物理位置（物理位置是指：块内偏
移和元组长度的）。如果元组被更新（逻辑上做删除标记，表示删除旧版本元组，并插入新版本元组），则t_
ctid记录的是新版本元组的物理位置

/* Fields below here must match MinimalTupleData! */
//两个重要的标志位，htup_details.h文件中定义了每个标志位可设置的标志
uint16      t_infomask2; /* number of attributes + various flags */
//低11位表示当前元组的属性个数。高5位用于包括用于HOT技术及元组可见性的标志位
uint16      t_infomask; /* various flag bits, see below */
//用于标识元组当前的状态。如元组是否具有OID、是否有空属性等，t_infomask的每一位对应不同的
状态，共21种状态（代码中查找“HEAP_HASNULL”等）如下所列
uint8      t_hoff; /* sizeof header incl. bitmap, padding */
//元组头的大小

/* ^ - 23 bytes - ^ */
bits8      t_bits[FLEXIBLE_ARRAY_MEMBER]; /* bitmap of NULLs */
//标识该元组的哪些字段为空
/* MORE DATA FOLLOWS AT END OF STRUCT */ //本结构后，将紧跟着数据
};

```

对于 t\_infomask 和 t\_infomask2，其主要组合值如下：

```

/*
 * information stored in t_infomask: //t_infomask
 */
#define HEAP_HASNULL      0x0001 /* has null attribute(s) */
#define HEAP_HASVARWIDTH  0x0002 /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL  0x0004 /* has external stored attribute(s) */
#define HEAP_HASOID       0x0008 /* has an object-id field */
#define HEAP_XMAX_KEYSHR_LOCK 0x0010 /* xmax is a key-shared locker */
#define HEAP_COMBOCID     0x0020 /* t_cid is a combo cid */
#define HEAP_XMAX_EXCL_LOCK 0x0040 /* xmax is exclusive locker */
#define HEAP_XMAX_LOCK_ONLY 0x0080 /* xmax, if valid, is only a locker */
...
/*
 * information stored in t_infomask2: //t_infomask2
 */
#define HEAP_NATTS_MASK    0x07FF /* 11 bits for number of attributes */
/* bits 0x1800 are available */
#define HEAP_KEYS_UPDATED 0x2000 /* tuple was updated and key cols modified,
or tuple deleted */

```

```
#define HEAP_HOT_UPDATED    0x4000    /* tuple was HOT-updated */
#define HEAP_ONLY_TUPLE    0x8000    /* this is heap-only tuple */
```

### 9.1.3 快照的类型

PostgreSQL 提供的快照有多种类型，分别为：

- ❑ Normal MVCC snapshots：数据库运行期间，在事务管理器的管理下，正常地执行用户的 SQL 语句，这时，为并发机制 MVCC 实现的事务事务快照，称为 Normal MVCC snapshot。
- ❑ Historic MVCC snapshots：用于 Hot-Standby 系统下在备机进行源自主机的事务的快照做恢复，这是主备复制技术中事务相关部分。

### 9.1.4 快照的管理

PostgreSQL 通过 GetSnapshotData() 函数获取快照。此函数遍历所有的进程，找出所有活动的进程中事务号是最小的赋值给 xmin、找出所有已经提交的事务号是最大的赋值给 xmax，目的是用 [xmin, xmax] 这样一个区间标识出本事务可见的事务区间。然后用 xmin 和 xmax 作为快照的左右边界赋值。所以这是获取快照的最基本的一个函数。

只是，在获取 [xmin, xmax] 这样一个区间的过程中，需要考虑特殊的系统进程 --VACUUM 进程的事务号不影响快照的获取。

快照获取之上层，需要根据 PostgreSQL 的需要，为上层区分出不同类型的快照，如图 9-1 所示，包含有多种不同类型的快照调用了 GetSnapshotData() 函数。各种不同的快照类型，主要如下：

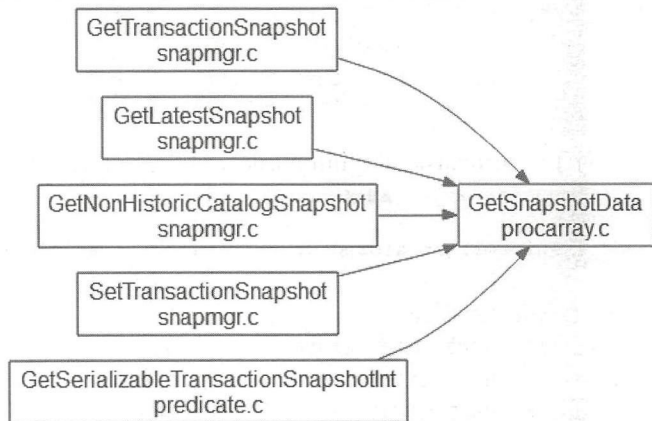


图 9-1 多种不同类型的快照图

- ❑ GetTransactionSnapshot()：

在可重复性读或可串行化隔离级别下获取事务的快照。处理可串行化隔离级别时，需要调用 GetSerializableTransactionSnapshotInt()。

- ❑ GetLatestSnapshot()：获取最新的快照。

- ❑ GetNonHistoricCatalogSnapshot()：获取系统表的最新的快照。

- ❑ GetSerializableTransactionSnapshotInt()：在可串行化隔离级别下获取事务的快照。

- ❑ SetTransactionSnapshot()：可以把数据库引擎之前保存的一个快照读入并设置为当前的快照。

例如，获取系统元数据（system catalog）快照的调用栈如下，可以看到，需要读取系统



表的时候，可以从系统表的快照来获取，以保持系统表数据获取的数据一致性。

```
main()
SubPostmasterMain()
BackendRun()
PostgresMain()
    exec_simple_query()
    pg_plan_queries()
    pg_plan_query()
    planner()
        standard_planner()
        subquery_planner()
        grouping_planner()
        query_planner()
            add_base_rels_to_query()
            add_base_rels_to_query()
            get_relation_info()
            RelationGetFKKeyList()
            systable_beginscan() //需要扫描系统表
            index_open()
            relation_open()
            RelationIdGetRelation()
            RelationBuildDesc()
            ScanPgRelation()
            GetCatalogSnapshot()
            //获取系统元数据(system catalog)的快照
            GetNonHistoricCatalogSnapshot()
            GetSnapshotData()
```

对于一个普通的事务，如执行一个普通的查询，则在查询重写（pg\_analyze\_and\_rewrite() 函数被调用之前）优化前，即已经建立好了快照，用于为查询计划的获取使用。但是，执行器在执行查询语句时（调用 PortalStart(Portal portal, ParamListInfo params, int eflags, Snapshot snapsho) 函数），第四个参数传入的值是“InvalidSnapshot”，表明执行器需要在执行期间重新获取快照。

再比如，对于 PREPARE 语句，在调用 EXECUTE 语句（调用 ExecuteQuery() 函数）真正执行语句时，才会建立快照，方式如下：

```
PortalStart(portal, paramLI, eflags, GetActiveSnapshot());
```

再比如，使用如下命令在设置约束时，调用 afterTriggerInvokeEvents() 函数时，需要保存快照的状态，之后恢复。

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

因此需要注意：快照的获取的时机，是不相同的，有着很多种情况，如图 9-2 所示，具体代码不再详述，请参见源码。





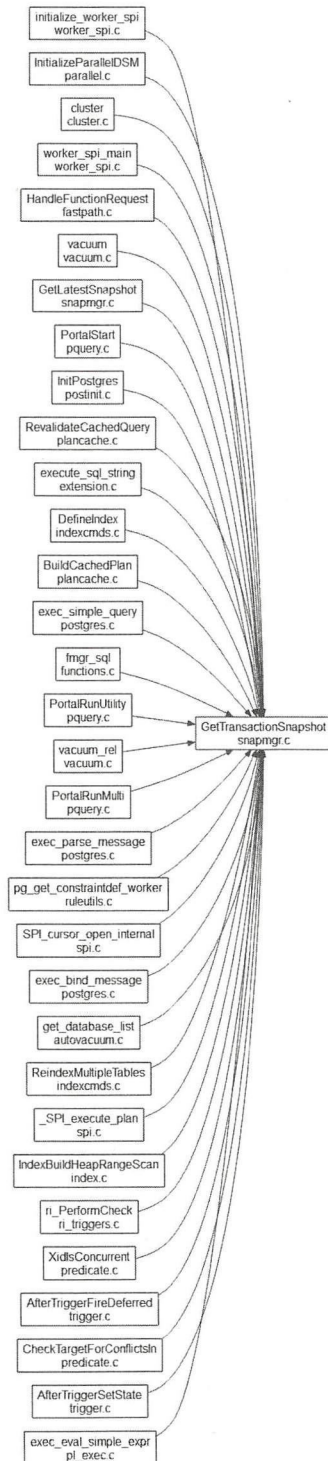


图 9-2 获取快照的上下文关系图



### 9.1.5 可串行化隔离级别的快照

可串行化快照的获取，实则通过 `GetSerializableTransactionSnapshotInt()` 函数完成的，但是，有个特例是“只读可延迟事务一定不会发生读写冲突”，所以即使设置了隔离级别是可串行化隔离级别，也不必在可串行化快照下运行，而是在普通的快照下运行也能保证数据的一致性，所以包装了一层函数。

```
Snapshot
GetSerializableTransactionSnapshot(Snapshot snapshot)
{...
    /*
     * A special optimization is available for SERIALIZABLE READ ONLY DEFERRABLE transactions
     * -- we can wait for a suitable snapshot and thereby avoid all SSI overhead
     * once it's running. */
    if (XactReadOnly && XactDeferrable)
        //如果事务是只读且是可延迟的，则一定不必在可串行化隔离级别下运行
        return GetSafeSnapshot(snapshot);
        //即使设置的隔离级别是可串行化隔离级别。即意味着只读可延迟事务一定不会发生读写冲突

    return GetSerializableTransactionSnapshotInt(snapshot, InvalidTransactionId);
    //否则，才获得一个可串行化事务快照
}
```

下面讨论 `GetSerializableTransactionSnapshotInt()` 函数的实现。主要有两个问题，一是获取快照，二是找出可能与本事务存在读写冲突的事务做出标记。

```
static Snapshot
GetSerializableTransactionSnapshotInt(Snapshot snapshot, TransactionId sourcexid)
{...
    LWLockAcquire(SerializableXactHashLock, LW_EXCLUSIVE);
    do
    {
        sxact = CreatePredXact();
        //从长⊖的一个列表中 (PredXact->availableList) 取出一个可以使用的可串行化事务
        /* If null, push out committed sxact to SLRU summary & retry. */
        if (!sxact) //如果定量的事务配额用光，则汇总一部分已经提交了的事务为一个事务
        {
            LWLockRelease(SerializableXactHashLock);
            SummarizeOldestCommittedSxact(); //汇总一部分已经提交了的事务为一个事务
            LWLockAcquire(SerializableXactHashLock, LW_EXCLUSIVE);
        }
    } while (!sxact);
}
```

⊖ PostgreSQL为了节约内存，防止谓词锁（SIREAD）等信息量太大耗费内存，使用了定量支持可串行化事务的个数、可串行化事务个数将要超出最大值时汇总一部分已经提交的可串行化事务信息到一个“dummy”事务中，化N个为1个，达到了节约内存的目的。





```

/* Get the snapshot, or check that it's safe to use */
if (!TransactionIdIsValid(sourcexid))
//对于可串行化快照的获取，本函数的入口参数传入的是“InvalidTransactionId”，
    snapshot = GetSnapshotData(snapshot); //意味着一定会调用GetSnapshotData()来
    获得一个物理快照，而此函数同样在可重复读隔离级别下被调用，表明可串行化隔离级别是比可重复读多的是SSI技术下的读写冲突检测而非快照的获取方式，由此体会这两个隔离级别实现的差异----即本段代码之前
    和之后所做的多种判断工作，正是可串行化隔离级别需要的必要的工作
    else if (!ProcArrayInstallImportedXmin(snapshot->xmin, sourcexid))
    {...}

if (XactReadOnly && PredXact->WritableSxactCount == 0) //如果本事务是只读的，且
    当前不存在写事务，则不可能构成读写重复，因此可不在可串行化隔离级别下运行，直接返回快照
{
    ReleasePredXact(sxact);
    LWLockRelease(SerializableXactHashLock);
    return snapshot;
}
...
/* Initialize the structure. */
sxact->vxid = vxid;
sxact->SeqNo.lastCommitBeforeSnapshot = PredXact->LastSxactCommitSeqNo;
...
if (XactReadOnly) //如果本事务是只读的，则判断是否可能存在读写冲突，判断条件如下
{
    sxact->flags |= SXACT_FLAG_READ_ONLY;
    /* Register all concurrent r/w transactions as possible conflicts;
     * if all of them commit without any outgoing conflicts to earlier transactions
     * then this snapshot can be deemed safe (and we can run without tracking
     * predicate locks). */
    for (othersxact = FirstPredXact(); othersxact != NULL; othersxact =
        NextPredXact(othersxact)) //遍历所有的活动事务
    {
        if (!SxactIsCommitted(othersxact) //活动的事务：即不是已经提交的
            && !SxactIsDoomed(othersxact) //活动的事务：也不是已经注定要回滚的
            && !SxactIsReadOnly(othersxact)) //活动的事务：还不是只读的，那么这样
            的事务就可能与本事务存在读写冲突
        {
            SetPossibleUnsafeConflict(sxact, othersxact); //记载可能的冲突关系，
            将来进一步由
        }
    }
}
else //本事务不是只读的
{
    ++(PredXact->WritableSxactCount); //对存在的写事务计数
    Assert(PredXact->WritableSxactCount <= (MaxBackends + max_prepared_xacts));
}

```





```
...  
    LocalPredicateLockHash = hash_create("Local predicate lock",  
    //创建本会话/进程独享的本地谓词锁的Hash表，便于将来快速搜索  
    max_predicate_locks_per_xact, &hash_ctl, HASH_ELEM | HASH_BLOBS);  
    return snapshot;  
}
```

## 9.2 可见性判断与多版本

PostgreSQL 的并发控制技术是以 MVCC 技术为主，封锁技术为辅。这一点和 MySQL 的 InnoDB 不同。InnoDB 是在以封锁技术为主体的情况下，用 MVCC 技术辅助实现读-写、写-读操作的并发。辨别一个数据库管理系统主要的并发控制技术，是看其实现使用什么技术实现可串行化。因为只有实现了可串行化，才能真正保证数据的一致性，这是功能正确性的问题。而隔离级别中除可串行化级别外的其他级别，是在牺牲一致性的前提下，从提高性能方面考虑的，所以越是弱的隔离级别，限制越少，并发度越高，所以除可串行化级别外的其他级别则是性能问题。先功能正确再追求性能卓越，是数据库管理系统各个功能特性设计的目标。PostgreSQL 是基于 MVCC 技术实现的可串行化（参见 9.3 和 9.4 节），所以我们说 PostgreSQL 的并发控制技术是以 MVCC 技术为主。而 InnoDB 是基于锁实现的可串行化，所以 InnoDB 的并发控制技术是以封锁技术为主。但 InnoDB 也支持 MVCC。

尽管 PostgreSQL 和 InnoDB 对于 MVCC 技术的使用方式不同，但是二者毕竟都使用了 MVCC 的多版本和快照技术，这意味着都存在多版本的可见性判断。如下就 PostgreSQL 的实现和代码就可见性和多版本进行讨论。

### 9.2.1 可见性判断

可见性判断，因各种操作对元组的获取需求不同而判断方式不同，有的操作需要读取到全部元组、有的操作则需要根据事务的特性为用户取到符合数据一致性要求的元组，所以 PostgreSQL 提供了多种可见性判断函数。

而通常意义的可见性判断，主要集中在事务的一致性和隔离级别的要求上，因此如下主要讨论事务中数据一致性前提下的元组可见性相关内容。但是，最后一小节“其他可见性判断”将概略性的讨论非事务相关的数据一致性下的数据可见性。

#### 1. 可见性判断依据

对于并发事务，如何读取某条元组的某个版本的方法，称为元组可见性判断。PostgreSQL 在元组上提供了四个系统字段（参见 9.1.2 节），用以标识元组的可见性，四个系统字段含义如下：



- ❑ xmin：标识首次创建这个版本的事务的时间戳值（事务号，xid），如使用 INSERT、COPY FROM 命令创建的元组的版本信息。
- ❑ xmax：标识被删除的时间戳值，如使用 DELETE 命令进行逻辑删除（VACUUM 执行物理删除）的事务的时间戳值。默认值为 0，是在创建时指定，即表示没有被删除。
- ❑ cmin、cmax：标识在一个事务块内部、多条语句命令的序列值。从 0 开始，用于同一个事务块中实现版本内的可见性判断，理同 xmin、xmax。cmin 对应的是创建类命令；cmax 对应的是删除类命令。
- ❑ 上面只涉及了版本的创建和删除，没有包括更新操作。对于更新操作，PostgreSQL 会“逻辑地删除”旧版本，“物理地生成”新版本，将当前事务 ID 即 xid 先存于旧版本的 xmax 中，再存于新版本的 xmin 中。
- ❑ 其他的用于判断可见性的还有如 9.1.2 节所述的 t\_infomask 和 t\_infomask2。

## 2. 快照可见性判断

PostgreSQL 利用快照和多版本技术，实现了 MVCC 技术与快照隔离。利用快照判断元组的可见性，详细的代码实现，不只是如上规则，还需要结合事务运行期间的动态状态、事务的执行过程来判断元组的可见性，所以更为复杂。

如下是元组对于指定的快照是否为此快照代表的事务可见的代码分析，分为三种情况进行判断，具体内容如下：

- ❑ 第一种情况：元组已经生成，但是还没有提交。这是插入操作正在进行中生成的元组。
- ❑ 第二种情况：元组已经生成，并提交，插入操作完成。
- ❑ 第三种情况：元组对应的事务不合法或被回滚。
- ❑ 第四种情况：元组被别的事务更新或删除，但这样的事务没有完成。
- ❑ 第五种情况：元组被别的事务更新或删除，这样的事务已经完成。

```
bool //元组的版本htup是否可以被参数snapshot所代表事务可见⊖？ 返回TRUE表示版本可见
HeapTupleSatisfiesMVCC(HeapTuple htup, Snapshot snapshot, Buffer buffer)
//注意，元组htup是谁生成的需要与快照的关系不大，除非是本事务自己生成的元组，需要和快照结合判断
本事务内对子事务的可见性
{...
    //第一种情况：元组已经生成，但是还没有提交。插入操作进行中。
    if (!HeapTupleHeaderXminCommitted(tuple)) //对于没有提交的元组，判断可见性。包括
        正在插入和正在更新的情况
    {
        if (HeapTupleHeaderXminInvalid(tuple)) //元组不合法，坏的元组
            return false;

        /* Used by pre-9.0 binary upgrades */
        if (tuple->t_infomask & HEAP_MOVED_OFF)
```

⊖ 注意这句话所要表明的主语和谓语，以及宾语。这样能更好地理解特定事务、快照、元组在可见性上之间的关系。





```

//适用于把9.0版本之前的二进制数据目录升级到9.0版本及之后的版本的数据
{...}
/* Used by pre-9.0 binary upgrades */
else if (tuple->t_infomask & HEAP_MOVED_IN) //同上
{...}
else if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetRawXmin(tuple)))
//如果是本事务生成的元组
{
    //本事务快照建立之后，本事务插入的数据，对于事务刚开始时即建立的快照而言，是不可见的
    if (HeapTupleHeaderGetCmin(tuple) >= snapshot->curcid)
    //但这种情况可能发生吗？本事务生成的数据对本事务可见，即读取本事务的数据是不需要快照
    //的。以，这只能发生在并发执行的多个子事务之间①
        return false; /* inserted after scan started */
    if (tuple->t_infomask & HEAP_XMAX_INVALID) /* xid invalid */
    //元组被更新或删除，但执行更新或删除的事务被回滚，则元组对于本事务而言是可见的
        return true;
    if (HEAP_XMAX_IS_LOCKED_ONLY(tuple->t_infomask)) /* not deleter */
    //元组只是被锁住，但还没有发生删除、修改等操作，故可见
        return true;
    if (tuple->t_infomask & HEAP_XMAX_IS_MULTI)
    //版本是其他事务执行更新操作修改过的（删除的旧版本）
    {
        //--元组是本事务生成的，但被其他事务更新。这可能发生吗？----
        //只能发生在并发的子事务间
        TransactionId xmax;
        xmax = HeapTupleGetUpdateXid(tuple);
        //取到这个版本的xmax（因为版本是其他事务执行更新操作生成的）
        ...

        /* updating subtransaction must have aborted */
        if (!TransactionIdIsCurrentTransactionId(xmax))
        //不是本事务删除此元组，生产此版本的更新子事务被回滚，则对于本事务是可见的
            return true;
        else if (HeapTupleHeaderGetCmax(tuple) >= snapshot->curcid)
        //更新是快照之后发生，则对于本事务是可见的
            return true; /* updated after scan started */
        else
            return false; /* updated before scan started */
        //更新是快照之前发生，则对于本事务是不可见的
    }

    if (!TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetRawXmax(tuple)))
    //执行删除操作的子事务被回滚，删除无效
    {
        /* deleting subtransaction must have aborted */
        SetHintBits(tuple, buffer, HEAP_XMAX_INVALID, InvalidTransactionId);
        return true; //执行删除操作的子事务被回滚，删除无效，故可见
    }
}

```

<sup>①</sup> PostgreSQL实现了“自治事务（Autonomous subtransactions）”。参见[https://wiki.postgresql.org/wiki/Autonomous\\_subtransactions](https://wiki.postgresql.org/wiki/Autonomous_subtransactions)。





```

        if (HeapTupleHeaderGetCmax(tuple) >= snapshot->curcid)
            return true;        /* deleted after scan started */
        //快照之后发生删除的元组, 对本事务可见
    else
        return false;        /* deleted before scan started */
        //快照之前发生删除的元组, 对本事务不可见
    }
    else if (XidInMVCCSnapshot(HeapTupleHeaderGetRawXmin(tuple), snapshot))
        //快照范围之内发生的, 不可见
        return false;
    else if (TransactionIdDidCommit(HeapTupleHeaderGetRawXmin(tuple)))
        SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED, HeapTupleHeaderGetRawXmin(tuple));
    //即不是本事务生产的数据, 也不在快照的范围内部, 则只有两种情况, 一是在快照的左边界之左面, 属于可见范围, 二是在快照的右边界之右面, 属于不可见范围。对于前者, 此处不做可见性判断, 留待如下借用“第三种情况”的代码进行可见性判断。对于后者, 对应的就是下面的else内容, 一定不可见
    else    //已被回滚或发生过crashed的元组, 对本事务不可见
    {
        /* it must have aborted or crashed */
        SetHintBits(tuple, buffer, HEAP_XMIN_INVALID, InvalidTransactionId);
        return false;        //二是在快照的右边界之右面, 属于不可见范围, 一定不可见
    }
}
else    //第二种情况: 元组已经生成, 已经提交
{
    /* xmin is committed, but maybe not according to our snapshot */
    //对于已经提交的元组, 判断可见性
    if (!HeapTupleHeaderXminFrozen(tuple) &&    //元组没有被冻结⊖ (已经提交的、不合法的才能被冻结), 所以结合上面的else进入的条件, 此处是在对没有冻结的元组进行判断
        XidInMVCCSnapshot(HeapTupleHeaderGetRawXmin(tuple), snapshot))
        //且元组快照范围之内发生的, 不可见
        return false;        /* treat as still in progress */
}
//第三种情况: 元组已经提交, 几乎应该都可见, 但是有例外情况
/* by here, the inserting transaction has committed */
//前面判断的是正在执行的, 所以正在执行的情况已经判断完毕
if (tuple->t_infomask & HEAP_XMAX_INVALID)    /* xid invalid or aborted */
    //已经提交且没有被删除, 则可见
    return true;
if (HEAP_XMAX_IS_LOCKED_ONLY(tuple->t_infomask))    //元组只是被锁住, 则可见
    return true;
//第四种情况: 元组被别的事务更新或删除, 但这样的事务没有完成
if (tuple->t_infomask & HEAP_XMAX_IS_MULTI)
    //此版本上发生了并发的更新或删除操作
    {...
        xmax = HeapTupleGetUpdateXid(tuple);
        ...
        if (TransactionIdIsCurrentTransactionId(xmax))

```

⊖ 事务ID由无符号的32位数表示, 事务ID的分配是采用ID递增的方式, 这样当事务ID用完时, 会出现wraparound问题。冻结事务表示被冻结的对象的事务ID是最古老的事务ID。





```

//更新或删除操作致使老版本置了xmax值, 判断是否是当前事务所为
{
    if (HeapTupleHeaderGetCmax(tuple) >= snapshot->curcid)
        //本事务内部, 后发生的删除操作对删除操作之前快照是可见的
        return true;    /* deleted after scan started */
    else
        return false;    /* deleted before scan started */
}
if (XidInMVCCSnapshot(xmax, snapshot)) //更新操作是当前事务所为, 且在快照范围内, 可见
    return true;
if (TransactionIdDidCommit(xmax))
    //更新操作是当前事务所为, 但已经提交, 则不可见 (更新操作是个子事务)
    return false;    /* updating transaction committed */
/* it must have aborted or crashed */
return true;
}

if (!(tuple->t_infomask & HEAP_XMAX_COMMITTED)) //此版本是最新版本, 没有被更新或删除
{
    if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetRawXmax(tuple)))
        //当前事务所为
        {
            if (HeapTupleHeaderGetCmax(tuple) >= snapshot->curcid)
                //本事务内部, 后发生的删除操作对删除操作之前快照是可见的
                return true;    /* deleted after scan started */
            else
                return false;    /* deleted before scan started */
        }

    if (XidInMVCCSnapshot(HeapTupleHeaderGetRawXmax(tuple), snapshot))
        //更新操作是当前事务所为, 且在快照范围内, 可见
        return true;

    if (!TransactionIdDidCommit(HeapTupleHeaderGetRawXmax(tuple)))
        //操作已经回滚, 则不可见 (更新操作是个子事务)
        {
            /* it must have aborted or crashed */
            SetHintBits(tuple, buffer, HEAP_XMAX_INVALID, InvalidTransactionId);
            return true;
        }

    /* xmax transaction committed */
    SetHintBits(tuple, buffer, HEAP_XMAX_COMMITTED, HeapTupleHeaderGetRawXmax(tuple));
}
else //第五种情况: 元组被别的事务更新或删除, 这样的事务已经完成。
{
    /* xmax is committed, but maybe not according to our snapshot */
    if (XidInMVCCSnapshot(HeapTupleHeaderGetRawXmax(tuple), snapshot))
        return true;    /* treat as still in progress */
}
}

```



```
/* xmax transaction committed */
return false;
}
```

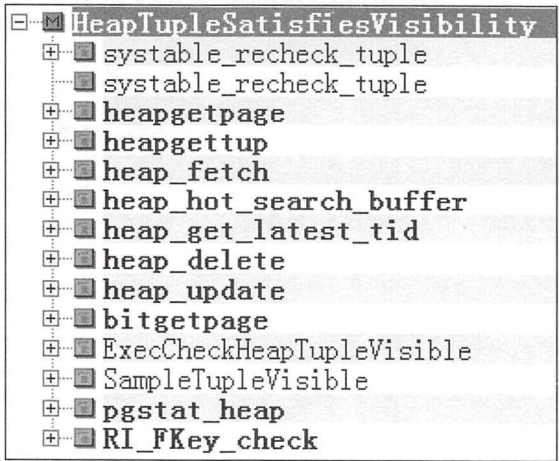


图 9-3 元组可见性判断上下文调用关系

3. 可见性判断示例

接下来，我们使用一个示例，来观察与元组可见性判断相关的内容，此示例以已提交读隔离级别为背景。

首先，数据准备如下：

```
CREATE TABLE test_tuple_versions (id INTEGER, value varchar(10));
```

其次，准备数据如下：

```
BEGIN;
INSERT INTO test_tuple_versions VALUES(1, 'a');
INSERT INTO test_tuple_versions VALUES(2, 'b'), (3, 'c');
COMMIT;
```

然后执行如下命令：

```
SELECT TXID_CURRENT(); //获得当前的事务号，结果得到的值为606，表明查询语句也会被分配事务号
SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;
```

得到的结果如下：

```
postgres=# SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;
 id | value | xmin | xmax | cmin | cmax | ctid
-----+-----+-----+-----+-----+-----+-----
  1 | a     | 605  |    0 |    0 |    0 | (0,1)
    //事务号为605的事务插入数据，所以xmin为605。ctid的值为(0,1)表示0页面第一个元组
  2 | b     | 605  |    0 |    1 |    1 | (0,2)
```





```
//同一个事务605的插入的数据。插入语句是本事务内的第二个命令，所以cmin、cmax为1
3 | c      | 605 | 0 | 1 | 1 | (0,3)、
//同上。ctid的值为(0,3)表示0页面第三个元组
(3 rows)
```

之后，分别执行两个并发事务，一个是 T1，一个是 T2，命令执行的顺序如表 9-3 所示。

表 9-3 并发事务执行表

时间	事务 T1	事务 T2
t0	BEGIN;	BEGIN
t1	UPDATE test_tuple_versions SET value = 'e' WHERE id = 2;	
t2		SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;
t3	SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;	
t4	COMMIT;	
t5		SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;

在时刻 t2 事务 T2 得到的查询结果如下：

```
postgres=# SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;
 id | value | xmin | xmax | cmin | cmax | ctid
-----+-----+-----+-----+-----+-----+-----
 1 | a      | 605 | 0 | 0 | 0 | (0,1)
//默认隔离级别是已提交读，所以尽管id为2的元组value值被改为e，但是看不到。而且，
 2 | b      | 605 | 609 | 0 | 0 | (0,2)
//ctid值为(0,2)表示看到的是物理位置为0页第二个元组，t3、t5时刻的查询表明这是个旧版本
 3 | c      | 605 | 0 | 1 | 1 | (0,3)
//但是，xmax被标为了609，表明事务609已经更新或删除了id值为2的元组
(3 rows)
```

在时刻 t3 事务 T1 得到的查询结果如下：

```
postgres=# SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;
 id | value | xmin | xmax | cmin | cmax | ctid
-----+-----+-----+-----+-----+-----+-----
 1 | a      | 605 | 0 | 0 | 0 | (0,1)
 3 | c      | 605 | 0 | 1 | 1 | (0,3)
//事务T1是属主，自己执行更新后，看不到旧版本了，只能看到ctid值为(0,4)的新版本
 2 | e      | 609 | 0 | 0 | 0 | (0,4)
//而且，新版本的xmin值是更新操作所在的事务的事务号
(3 rows)
```

在时刻 t5 事务 T2 得到的查询结果如下：

```
postgres=# SELECT *, xmin, xmax, cmin, cmax, ctid FROM test_tuple_versions;
```

```
id | value | xmin | xmax | cmin | cmax | ctid
---+-----+-----+-----+-----+-----+-----
1 | a     | 605 | 0 | 0 | 0 | (0,1)
3 | c     | 605 | 0 | 1 | 1 | (0,3)
2 | e     | 609 | 0 | 0 | 0 | (0,4)

//默认隔离级别是读已提交，因事务T1已经提交，所以可以看到新版本的状态
(3 rows)
```

读者可以自行尝试一个事务块内增删改查等操作对 cmin 和 cmax 等系统列的影响。

4. 更新可见性判断

HeapTupleSatisfiesUpdate() 函数是更新操作对所读到的元组的可见性判断函数，其实现基本山类似前述的 HeapTupleSatisfiesMVCC() 函数，但是，也有不同，如下是这两个函数实现的对比图，如图 9-4 所示是函数的初始部分，多数代码相似即判断的逻辑一致，只是返回值不同，这是因为更新操作需要对是否可见做更细粒度的判断。

HeapTupleSatisfiesUpdate	HeapTupleSatisfiesMVCC
1 HTSU Result	1 bool
2 HeapTupleSatisfiesUpdate(HeapTuple htup, CommandId curcid,	2 HeapTupleSatisfiesMVCC(HeapTuple htup, Snapshot snapshot,
3     Buffer buffer)	3     Buffer buffer)
4 {	4 {
5     HeapTupleHeader tuple = htup->t_data;	5     HeapTupleHeader tuple = htup->t_data;
6	6
7     Assert(ItemPointerIsValid(&htup->t_self));	7     Assert(ItemPointerIsValid(&htup->t_self));
8     Assert(htup->t_tableOid != InvalidOid);	8     Assert(htup->t_tableOid != InvalidOid);
9	9
10    if (!HeapTupleHeaderXminCommitted(tuple))	10    if (!HeapTupleHeaderXminCommitted(tuple))
11    {	11    {
12      if (HeapTupleHeaderXminInvalid(tuple))	12      if (HeapTupleHeaderXminInvalid(tuple))
13        return HeapTupleInvisible;	13        return false;
14	14
15    /* Used by pre-9.0 binary upgrades */	15    /* Used by pre-9.0 binary upgrades */
16    if (tuple->t_infomask & HEAP_MOVED_OFF)	16    if (tuple->t_infomask & HEAP_MOVED_OFF)
17    {	17    {
18      TransactionId xvac = HeapTupleHeaderGetXvac(tuple);	18      TransactionId xvac = HeapTupleHeaderGetXvac(tuple);
19	19
20      if (TransactionIdIsCurrentTransactionId(xvac))	20      if (TransactionIdIsCurrentTransactionId(xvac))
21        return HeapTupleInvisible;	21        return false;
22      if (!TransactionIdIsInProgress(xvac))	22      if (!XidInMVCCSnapshot(xvac, snapshot))
23      {	23      {
24        if (TransactionIdDidCommit(xvac))	24        if (TransactionIdDidCommit(xvac))
25        {	25        {
26          SetHintBits(tuple, buffer, HEAP_XMIN_INVALID,	26          SetHintBits(tuple, buffer, HEAP_XMIN_INVALID,
27            InvalidTransactionId);	27            InvalidTransactionId);
28          return HeapTupleInvisible;	28          return false;
29        }	29        }
30        SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED,	30        SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED,
31          InvalidTransactionId);	31          InvalidTransactionId);
32        }	32        }
33      }	33      }
34    /* Used by pre-9.0 binary upgrades */	34    /* Used by pre-9.0 binary upgrades */
35    else if (tuple->t_infomask & HEAP_MOVED_IN)	35    else if (tuple->t_infomask & HEAP_MOVED_IN)
36    {	36    {
37      TransactionId xvac = HeapTupleHeaderGetXvac(tuple);	37      TransactionId xvac = HeapTupleHeaderGetXvac(tuple);
38	38
39      if (!TransactionIdIsCurrentTransactionId(xvac))	39      if (!TransactionIdIsCurrentTransactionId(xvac))
40      {	40      {
41        if (TransactionIdIsInProgress(xvac))	41        if (XidInMVCCSnapshot(xvac, snapshot))
42          return HeapTupleInvisible;	42          return false;
43        if (TransactionIdDidCommit(xvac))	43        if (TransactionIdDidCommit(xvac))
44          SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED,	44          SetHintBits(tuple, buffer, HEAP_XMIN_COMMITTED,
45            InvalidTransactionId);	45            InvalidTransactionId);
46        else	46        else
47        {	47        {
48          SetHintBits(tuple, buffer, HEAP_XMIN_INVALID,	48          SetHintBits(tuple, buffer, HEAP_XMIN_INVALID,
49            InvalidTransactionId);	49            InvalidTransactionId);
50          return HeapTupleInvisible;	50          return false;

图 9-4 更新操作的可见性判断对比图（一）

如图 9-5 所示是 HeapTupleSatisfiesUpdate() 函数的中段代码的对比，比较的是元组是快照



对应的事务块内生成、且判断快照事务（实则是并发的子事务间的可见性，如果是单一事务块，则本事务块内生成的元组对本事务都是可见的）对元组是否可见的情况，这时的处理就有较大的不同。通过对比可掌握更新操作的可见性判断原则，详细情况请读者自行对比掌握。

HeapTupleSatisfiesUpdate	HeapTupleSatisfiesMVCC
<pre> 53 } 54 else if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetRawXmin(tuple))) 55 { 56     if (HeapTupleHeaderGetCmin(tuple) &gt;= curcid) 57         return HeapTupleInvisible; /* inserted after scan started */ 58 59     if (tuple-&gt;t_infomask &amp; HEAP_XMAX_INVALID) /* xid invalid */ 60         return HeapTupleMayBeUpdated; 61 62     if (HEAP_XMAX_IS_LOCKED_ONLY(tuple-&gt;t_infomask)) 63     { 64         TransactionId xmax; 65 66         xmax = HeapTupleHeaderGetRawXmax(tuple); 67 68         /* 69          * Careful here: even though this tuple was created by our own 70          * transaction, it might be locked by other transactions, if 71          * the original version was key-share locked when we updated 72          * it. 73          */ 74 75         if (tuple-&gt;t_infomask &amp; HEAP_XMAX_IS_MULTI) 76         { 77             if (MultiXactIsRunning(xmax, true)) 78                 return HeapTupleBeingUpdated; 79             else 80                 return HeapTupleMayBeUpdated; 81         } 82 83         /* 84          * If the locker is gone, then there is nothing of interest 85          * left in this Xmax; otherwise, report the tuple as 86          * locked/updated. 87          */ 88         if (!TransactionIdIsInProgress(xmax)) 89             return HeapTupleMayBeUpdated; 90         return HeapTupleBeingUpdated; 91     } 92 93     if (tuple-&gt;t_infomask &amp; HEAP_XMAX_IS_MULTI) 94     { 95         TransactionId xmax; 96 97         xmax = HeapTupleGetUpdateXid(tuple); 98 99         /* not LOCKED_ONLY, so it has to have an xmax */ 100         Assert(TransactionIdIsValid(xmax)); 101 102         /* deleting subtransaction must have aborted */ 103         if (!TransactionIdIsCurrentTransactionId(xmax)) 104         { 105             if (MultiXactIsRunning(HeapTupleHeaderGetRawXmax(tuple), 106                                     false)) </pre>	<pre> 53 } 54 else if (TransactionIdIsCurrentTransactionId(HeapTupleHeaderGetRawXmin(tuple))) 55 { 56     if (HeapTupleHeaderGetCmin(tuple) &gt;= snapshot-&gt;curcid) 57         return false; /* inserted after scan started */ 58 59     if (tuple-&gt;t_infomask &amp; HEAP_XMAX_INVALID) /* xid invalid */ 60         return true; 61 62     if (HEAP_XMAX_IS_LOCKED_ONLY(tuple-&gt;t_infomask)) /* not deleter */ 63         return true; 64 65     if (tuple-&gt;t_infomask &amp; HEAP_XMAX_IS_MULTI) 66     { 67         TransactionId xmax; 68 69         xmax = HeapTupleGetUpdateXid(tuple); 70 71         /* not LOCKED_ONLY, so it has to have an xmax */ 72         Assert(TransactionIdIsValid(xmax)); 73 74         /* updating subtransaction must have aborted */ 75         if (!TransactionIdIsCurrentTransactionId(xmax)) 76             return true; 77         else if (HeapTupleHeaderGetCmax(tuple) &gt;= snapshot-&gt;curcid) 78             return true; /* updated after scan started */ </pre>

图 9-5 更新操作的可见性判断对比图（二）

## 5. 其他可见性判断

宏观意义上的可见性判断相关函数，如表 9-4 所示。这些函数分别从用户事务视角、用户非事务视角、系统内部操作视角等多个角度来获取元组，因而有不同的实现方式。实现细节不再详述，请参见相关代码。

表 9-4 所有的元组可见性判断函数

可见性判断函数	功能说明
HeapTupleSatisfiesMVCC() 用户事务视角	visible to supplied snapshot, excludes current command 用于 MVCC 技术实现快照的事务隔离级别。主要是读取数据时判断元组可见性，而读取数据又有多种方式，如全表扫描、索引扫描、位图扫描、布隆过滤扫描等多种扫描方式
HeapTupleSatisfiesUpdate() 用户事务视角	visible to instant snapshot, with user-supplied command counter and more complex result 用于更新或删除操作中对元组的可见性进行判断



(续)

可见性判断函数	功能说明
HeapTupleSatisfiesSelf() 用户事务视角	visible to instant snapshot and current command 多用于 contrib 中“强制标记安全访问”特性 (sepgsql 模块), 判断元组的可见性
HeapTupleSatisfiesDirty() 系统内部操作视角	like HeapTupleSatisfiesSelf(), but includes open transactions 扫描系统表, 不能受隔离级别控制, 需要获得最新的信息, 因而类似脏读的方式获取系统表中的数据。例如检查约束、为用户返回元组的信息方便了解系统状态 (如 pgstat_relation() 函数) 等
HeapTupleSatisfiesVacuum() 系统内部操作视角	visible to any running transaction, used by VACUUM 为 VACUUM、ANALYZE 等操作扫描元组时确定哪些元组可见
HeapTupleSatisfiesToast() 用户事务视角	visible unless part of interrupted vacuum, used for TOAST 对 TOAST 数据的插入、删除、修改等操作进行时, 判断对 TOAST 数据的可见性
HeapTupleSatisfiesAny() 用户非事务视角	all tuples are visible 所有的元组都是可见的, 适合做数据统计、通过 copy_heap_data() 函数做数据的物理拷贝如实现聚集索引时需要移动全部元组数据等操作

9.2.2 多版本实现

多版本, 是指每个元组对象, 有多个不同的版本, 除第一个版本是插入操作生成外, 其他的多个版本, 是多次更新操作生产的。

1. 版本结构

多个版本之间, 使用元组头 HeapTupleHeaderData 中的 t\_ctid 作为“指针”指向下一个版本, 而这样的指针, 是一个物理地址, 即下一个版本在哪个页面的哪个偏移处。

对于 t\_ctid, 其基本结构如下:

```
typedef struct ItemPointerData
{
    BlockIdData ip_blkid;    //32位的块地址, 分为bi_hi和bi_lo两部分, 表示一个物理页面的地址
    OffsetNumber ip_posid;    //16位的无符号数, 表示在一个页面内部的偏移量
}
```

如果是只有一个版本 (只发生过插入操作), 则 t\_ctid 的值就是本条元组的第一个版本的物理地址, 即指向自己。如果存在多个版本, 则最后一个版本的 t\_ctid 也指向此版本自己。

2. 版本生成

PostgreSQL 中对于元组采用多版本技术存储。

在 V8.3 之前, 对元组的插入操作产生第一个版本, 之后, 每个更新操作都会产生一个新版本 (通过 HeapTupleHeaderSetCmin() 宏为 t\_cid 赋值, 加载插入操作的命令 ID)。多个版本之间从旧到新版本, 将旧版本的 t\_ctid 字段指向下一个版本的位置, 这样逐个指向, 就可形成一条版本链。

另外，更新操作不但会在物理页面上产生元组的新版本，还会在索引页中也产生新版本的索引记录。但是，当更新操作没有修改索引属性，PostgreSQL也会对每个索引项产生一个新版本。显然这会浪费存储空间，旧的版本只有在执行 VACUUM 时才能被回收。

为了解决如上的问题，从 V8.3 开始，PostgreSQL 使用了一种称为 HOT 机制的技术。当更新的元组同时满足如下条件时（通过 HeapSatisfiesHOTUpdate() 函数判断）称为 HOT 元组（heap-only tuple，意思是只是 heap 中的内容，不是元组的全部物理存储内容，索引项通常在 heap 中但不意味着 heap 中只有索引项）：

- ❑ 索引属性没被修改：所有的索引项上的每一个值都没有被修改，即只修改了数据没有修改索引。一个特例是，如果一条 UPDATE 语句修改了某属性，但前后值相同则认为没有修改。
- ❑ 元组新版本与旧版本在同一个页面内：在同一个页面内查找方便，不会引发可能的 IO 操作。如果不满足上述条件，则还是采用老办法生成元组的新版本。

当找到 HOT 元组后，这样的元组就会被设置 HEAP\_ONLY\_TUPLE 标志（HeapTupleSetHeapOnly() 宏），而 HOT 元组的上一个版本则被设置 HEAP\_HOT\_UPDATED 标志（HeapTupleSetHotUpdated() 宏）。这种情况下，更新一条 HOT 元组不再会在索引中引入新版本的索引项，即不会增加索引的大小。

如果通过索引获取元组，则会找到同一页面中最老的版本，然后顺着版本链向后找（根据当前元组的老版本上的 t\_ctid 找下一个版本），直到根据快照技术找到满足元组可见性的元组（参见 9.2 节），即元组对于本事务是可见的。

如果更新操作的对象是本事务插入的元组，则不生成新的版本。

### 3. 版本查找

在版本链上做查找，有三个关键点：一是要循着 t\_ctid 找元组的下一个版本，二是要对找到的每个版本做可见性判断（即符合隔离级别的要求），三是要进一步判断可串行化隔离级别下是否构成读写冲突（参见 9.3、9.4、9.5 节）。heap\_get\_latest\_tid() 函数典型地实现了这三个关键点。基于索引、位图等方式做版本的查找，则不需要循着 t\_ctid 找元组的下一个版本，但是做可见性判断和读写冲突检查是必要的，如 bitgetpage() 函数。

```
void //在指定的表(relation)中，根据快照查找元组，如果存在多个版本，则根据每个版本的
    t_ctid找下一条版本
heap_get_latest_tid(Relation relation, Snapshot snapshot, ItemPointer tid)
{...
    ctid = *tid;
    priorXmax = InvalidTransactionId;    /* cannot check first XMIN */
    for (;;) //无限循环
    {...
        buffer = ReadBuffer(relation, ItemPointerGetBlockNumber(&ctid));
        //获取页面的物理地址、元组的偏移地址等
```



```

LockBuffer(buffer, BUFFER_LOCK_SHARE);
page = BufferGetPage(buffer);

...

offnum = ItemPointerGetOffsetNumber(&ctid);

...

/* OK to access the tuple */
tp.t_self = ctid; //构造元组的信息, 准备获取这样的元组的其他信息、如t_xmin、t_xmax等以判断元组的可见性等
tp.t_data = (HeapTupleHeader) PageGetItem(page, lp);
tp.t_len = ItemIdGetLength(lp);
tp.t_tableOid = RelationGetRelid(relation);

...

valid = HeapTupleSatisfiesVisibility(&tp, snapshot, buffer);
//根据快照判断元组的可见性
CheckForSerializableConflictOut(valid, relation, &tp, buffer, snapshot);
//根据SSI技术做读写冲突检查
if (valid) //如果合法
    *tid = ctid;
if ((tp.t_data->t_infomask & HEAP_XMAX_INVALID) ||
//元组不合法, 要么是被回滚, 要么是系统Crashed, 不应该再找下去
    HeapTupleHeaderIsOnlyLocked(tp.t_data) ||
//对元组所处的状态进行判断, 如是否提交是否加锁等
    ItemPointerEquals(&tp.t_self, &tp.t_data->t_ctid))
//如果t_ctid指向本元组自己, 则找到, 可以不再寻找
{
    UnlockReleaseBuffer(buffer);

    break; //退出循环
}

ctid = tp.t_data->t_ctid; //上面条件如果不满足, 则找版本链条中的下一条元组,
//而下一条元组是通过本元组的t_ctid指向的
priorXmax = HeapTupleHeaderGetUpdateXid(tp.t_data);
UnlockReleaseBuffer(buffer);
} /* end of loop */
}

```

#### 4. 版本清理

PostgreSQL 删除一个元组的时候, 通过 `HeapTupleHeaderSetCmax()` 宏, 要把删除操作的事务号赋值给 `t_xmax`, 把删除命令的命令 ID 赋值给 `t_cid`, 为一个版本画上生命结束的句号。

之后, 系统的 `VACUUM` 进程, 将扫描页面, 清理无用的版本。



## 9.3 可串行化快照原理

PostgreSQL 支持 MVCC 技术，但是 MVCC 技术会带来写偏序（Write Skew）的问题（参见 1.2 节）。写偏序异常在 PostgreSQL 的早期版本中存在，但是 PostgreSQL V9.2 版本解决了这个问题。

PostgreSQL 通过采用可串行化快照隔离技术（Serializable Snapshot Isolation，SSI），解决了写偏序异常。这一节将重点讨论 SSI 技术的原理。

### 9.3.1 理论基础

在论文《Generalized Isolation Level Definitions》中，根据事务对数据项（同一个数据项 X）的读或写操作，定义了事务之间的三种关系，如图 9-6 所示。这三种关系，分别是：

- ❑ **写依赖（ww-dependency）**：写-写依赖关系，即两个写操作修改同一个数据项，分别生成不同的版本，先生成版本的事务指向后生成版本的事务，“指向”即表示依赖，被指向的是被依赖者，如图 9-6，事务  $T_i$  指向  $T_j$ ，就是在写-写操作并发时，事务  $T_i$  依赖  $T_j$ （因为最终以  $T_j$  生成的版本为主，**依赖新值**）。
- ❑ **读依赖（wr-dependency）**：写-读依赖关系，事务  $T_i$  先写了一个版本  $X_i$ ，后发生的事务  $T_j$  读这个版本或者在谓词条件作用下读这个数据项 X，而  $T_i$  写的版本  $X_i$  影响了事务  $T_i$  读取的结果集，如图 9-6，事务  $T_i$  指向  $T_j$ ，表示读写操作之间存在依赖关系。
- ❑ **反依赖（rw-antidependency）**：读-写反依赖，简单表述为**读-写依赖**，即：先发生的读操作读取了版本  $X_i$ ，后发生的写事务  $T_j$  写了一个新版本，数据项 X 的值因而被改变，而  $T_j$  写的版本  $X_j$  不在事务  $T_i$  读取的结果集内，如图 9-6，事务  $T_i$  指向  $T_j$ ，表示读写操作之间存在与“写-读依赖关系”反向的依赖关系，此关系是先读后写。不**依赖新值**，而是**依赖新值之前的旧版本**的值，被称为了反依赖（或理解为**依赖旧值**）。

Conflicts Name	Description ( $T_j$ conflicts on $T_i$ )	Notation in DSG
写依赖 Directly write-depends	$T_i$ installs $x_i$ and $T_j$ installs $x$ 's next version	$T_i \xrightarrow{ww} T_j$
读依赖 Directly read-depends	$T_i$ installs $x_i$ , $T_j$ reads $x_i$ or $T_j$ performs a predicate-based read, $x_i$ changes the matches of $T_j$ 's read, and $x_i$ is the same or an earlier version of $x$ in $T_j$ 's read	$T_i \xrightarrow{wr} T_j$
反依赖 Directly anti-depends	$T_i$ reads $x_h$ and $T_j$ installs $x$ 's next version or $T_i$ performs a predicate-based read and $T_j$ overwrites this read	$T_i \xrightarrow{rw} T_j$

Figure 2. Definitions of direct conflicts between transactions.

图 9-6 并发事务之间的关联关系图

首先，这对应了读写操作的四种组合中的存在冲突的三种（参见 1.1.3 节），完全和第 1

章讲述的内容吻合。但这只是三种分类。

其次，三种依赖关系，也对应了 1.2.2 节讲述的“冲突行为”，表明数据库要解决的正是这三种冲突（读-读不冲突），这是冲突发生的根源。

再次，依据上述内容，就可以在并发事务之间，根据事务之间的三种关系，画出一幅有向事务关系图，表明事务操作数据项时的前后关系和读写操作对新版本值的依赖关系。这样的图，被称为 Direct Serialization Graph，简称为 DSG 图，是一个有向图。在此有向图中，事务是节点，事务间的关系是边，如上表述的三种依赖关系就是三种有向边；如果 DSG 图中存在环，则有写偏序异常，表明事务间的调度是非可串行化的。典型的两种环的情况如图 1-1 和图 1-2 所示。

第四，从事务 Tj 到 Tk 如果是写-读依赖（wr-dependency）则事务 Tj 一定在事务 Tk 之前提交，否则根据多版本的数据可见性判断，Tk 是不能读到事务 Tj 所写的数据的，即不可能构成写-读依赖的。

第五，而从事务 Tj 到 Tk 如果是写-写依赖（ww-dependency）因为写锁会排斥写锁，所以只能是 Tj 先提交才能构成写-写依赖关系。

第六，第四和第五合起来，表明写-读依赖和写-写依赖一定不是并发间的事务可能造成的。而读-写依赖则是并发的的事务间才可能发生的。

在 1.1.5 节，我们讨论了写偏序异常（Write Skew），并提出如下内容：

快照隔离并发控制技术的缺点，是并不能真正保证事务为“可序列化的”，即事务间的并发操作依旧会引发数据异常现象，但是这里的数据异常现象区别于前面提到的各种异常现象，其异常现象是“业务的逻辑语义”引发的，即除了抽象的读写操作，数据间还应该满足一定语义，即约束（constraint）而写偏序这样的异常现象，可以在 DSG 图中以图的形式得以体现（形式化），包括两点内容，且分别在论文中被论述证明，成为解决写偏序的依据，内容如下：

- 《A Generalized Theory and Optimistic Implementations for Distributed Transactions》发现在快照隔离技术下每个 DSG 图中如果存在环则至少两条“读-写反依赖”的边存在。
- 《Making snapshot isolation serializable》则论述了写偏序发生时，两条“读-写反依赖”的边是相邻的。

这两条，成为了解决快照隔离技术带来的写偏序问题的理论基础。后面章节讲述的检测算法，不是在 DSG 图中找环，而是找一个节点，观察这个节点的两条边，一条入边（别的事务依赖自己而指向自己）和一条出边（自己依赖别人而指向被依赖者）。如果存在这样的边，则正符合上述的两篇论文论述的情况，即“可能”导致环出现，所以这个节点表示的事务将被回滚，以消除可能造成的写偏序异常。当然，这种方式不精准，存在误杀的情况，导致有些不该回滚的事务被回滚掉，加大了整个系统的事务回滚率。但是，在 Berkeley DB 和 PostgreSQL 中基于此方式实现的序列化快照技术，被实战证明是高效的一种实现方式。

接下来的两节里提到的论文，分别给出了测试数据以表明整体效率，相关数据请参考论文。

### 9.3.2 算法实现

在《Serializable Isolation for Snapshot Databases》这篇论文中，提出了解决写偏序的具体算法，并在 Berkeley DB 中加以实现。论文中把所使用到的有向图称为 Multiversion Serialization Graph，简称 MVSG，但本质上就是一个 DSG 图。论文表述：只修改了 692 行代码，即非常高效地解决了写偏序异常问题。

- 首先，在事务开始的时候，为一个事务的入边和出边设置初值。初值为 FALSE 表明不存在依赖关系。如图 9-7 所示。

**modified begin(T):**

```
existing SI code for begin(T)
set T.inConflict = T.outConflict = false
```

图 9-7 对开始事务时的修改

- $T.inConflict$ <sup>①</sup>：表示有一个读-写依赖（rw-dependency）从其他某个并发的事务指向当前事务 T。“inConflict”是事务 T 的入边，但却是其他某个并发的事务的出边的语义。
- $T.outConflict$ ：表示有一个读-写依赖（rw-dependency）从当前事务 T 指向其他某个并发的事务。“outConflict”是事务 T 的出边，但却是其他某个并发的事务的入边的语义。

- 其次，读数据项的时候，引入一个新的、但不与任何锁冲突的“SIREAD”锁。如图 9-8 所示。

**modified read(T, x):**

```
get lock(key=x, owner=T, mode=SIREAD)
if there is a WRITE lock(wl) on x
    set wl.owner.inConflict = true
    set T.outConflict = true

existing SI code for read(T, x)

for each version (xNew) of x
    that is newer than what T read:
        if xNew.creator is committed
            and xNew.creator.outConflict:
                abort(T)
                return UNSAFE_ERROR
    set xNew.creator.inConflict = true
    set T.outConflict = true
```

图 9-8 读操作时的修改

- “SIREAD”锁，这种锁在读数据的时候被施加，而快照隔离技术在读数据的时候不加锁。其语义是：以快照方式读取过此数据的某个版本。

■ “SIREAD”锁的特色是不阻塞任何锁。

■ 如果一个数据项上同时存在“SIREAD”锁和写锁，表明存在读-写依赖（rw-dependency）。

- 如果读取的数据项 X 上存在一个写锁，则施加写锁的事务与此读操作的当前事务存在一个读-写依赖（rw-dependency，先读后写，正符合此时读操作所做的判断），即当前事务 T 指向施加写锁的事务，所以需要给数据项 X 上的施加写锁事

① 论文原文：T.inConflict indicates whether or not there is a rw-dependency from a concurrent transaction to T, and T.outConflict indicates whether there is a rw-dependency from T to a concurrent transaction.



务设置入边为 TRUE，给当前事务 T 的出边设置为 TRUE。

○ 紧接着，沿用快照隔离时的读操作逻辑不变。

○ 遍历数据项 X 上的每一个更新的版本，即从当前事务出发，遍历每一个新版本。

■ 如果存在更新的版本对应的事务已经被提交的情况，且这个事务的出边是 TRUE，表明有连续的两个读-写依赖存在，这时就可以回滚事务 T，目的是确保可串行化。这种情况对应的是：T1 → T2 → T3 中的 T1 被回滚，T2 是已经被提交的更新的版本对应的事务。

■ 如果不存在上述情况，则更新的版本对应的事务的入边是 TRUE，被事务 T 指向；而事务 T 的出边则应设置为 TRUE。

○ 注意本事务发生在读操作时刻。所以只可能构成读-写依赖，意味着事务 T 只可能指向其他并发的

□ 第三，写数据项的时候，施加写锁。如图 9-9 所示。

○ 发现对象上存在“SIREAD”锁，且加“SIREAD”锁的事务正在运行，或者加“SIREAD”锁的事务提交时间比本事务的开始时间早，表明历史上发生过读操作，则：

■ 如果加“SIREAD”锁的事务已经提交，且此事务有入边，则回滚本事务 T。这种情况对应的是：T1 → T2 → T3 中的 T3 被回滚，T2 是已经提交的加“SIREAD”锁的事务。

■ 如果上述条件不成立，则加“SIREAD”锁的事务的出边和本事务 T 的入边都被设置为 TRUE，这表面两者之间构成了一个读-写依赖，由加“SIREAD”锁的事务指向本事务 T。

○ 紧接着，沿用快照隔离时的写操作逻辑不变。

○ 注意本事务发生在写操作时刻。所以读-写依赖情况下，只可能由其他并发的

□ 第四，事务提交时，如图 9-10 所示。

○ 如果事务 T 的入边和出边都是 TRUE，则回滚本事务 T。这种

**modified write(T, x, xNew):**

```
get lock(key=x, locker=T, mode=WRITE)

if there is a SIREAD lock(rl) on x
with rl.owner is running
or commit(rl.owner) > begin(T):
    if rl.owner is committed
    and rl.owner.inConflict:
        abort(T)
        return UNSAFE_ERROR
    set rl.owner.outConflict = true
    set T.inConflict = true

existing SI code for write(T, x, xNew)
# do not get WRITE lock again
```

图 9-9 写操作时的修改

**modified commit(T):**

```
if T.inConflict and T.outConflict:
    abort(T)
    return UNSAFE_ERROR

existing SI code for commit(T)
# release WRITE locks held by T
# but do not release SIREAD locks
```

图 9-10 提交事务时修改

情况对应的是： $T1 \rightarrow T2 \rightarrow T3$  中的  $T2$  被回滚， $T2$  是一个位于中间的“轴”。

- 紧接着，沿用快照隔离时的事务提交操作逻辑不变。
- 释放事务  $T$  持有的写锁。
- 如本事务持有“SIREAD”锁，不释放。
- “SIREAD”锁只有在所有与本事务并发发生的事务完成后，才能被释放（对应的是写偏序异常中的三个事务病法引发异常的情况，如图 1-2 所示）。

## 9.4 PostgreSQL可串行化快照的实现

PostgreSQL 对于可串行化快照技术的实现，基于《Serializable Snapshot Isolation in PostgreSQL》<sup>①</sup> 论文，这篇论文不仅讲述了可串行化快照的理论基础、PostgreSQL 对于 SSI 技术的实现方式，还提出了为支持只读事务而实现的“Safe Snapshots”“Deferable Transaction”，因为避免读写冲突造成事务回滚的影响而对被回滚的事务采取“safe retry”策略，以及两阶段提交对选取回滚读写冲突的事务的影响等重要话题，本节将结合这篇论文和代码按照论文讨论的内容顺序展开，逐步介绍 PostgreSQL 对 SSI 技术的实现方式。

### 9.4.1 PostgreSQL 的状况

PostgreSQL 9.1 之前的版本，支持 SI 即快照隔离，而快照隔离不是一个可串行化的事务管理机制，因为支持快照且读数据不加锁（读数据加锁是传统的 S2PL 技术实现序列化的关键），所以尽管 PostgreSQL 避免了三种读异常（参见 1.1.3 节）但没有实现可串行化（体现在出现了写偏序异常）。换句话说，早期的 PostgreSQL 版本是不支持可串行化隔离级别的，不能真正保证数据的一致性，可能出现的写偏序异常在 1.1.5 节讲述过。

对于 SI 的实现，PostgreSQL 首先是利用了一个私有的快照，在此快照下读取特定版本的数据，所以利用了多版本技术；而在数据项上禁止写并发更新，是通过元组锁实现的。

而 PostgreSQL 9.1 及之后的版本，支持 SSI 即可串行化快照隔离，才真正实现了可串行化隔离级别（读数据加锁是传统的 S2PL 技术实现可串行化的关键，PostgreSQL 不是做不到这一点而是此技术的并发度最低，导致数据库的性能低下），不仅避免了写偏序异常，且不是特别影响事务处理的性能，这一点，在 PostgreSQL 的发展历史上，有着重要意义。

SSI 技术是本节的重点，详细内容参见后续章节。接下来先看 PostgreSQL 实现 SSI 所基于的原理。

### 9.4.2 PostgreSQL 实现 SSI 的理论基础

注意本节讨论的范围，限制在可串行化一词限定的范围内。

<sup>①</sup> 这篇论文是理解 PostgreSQL 事务管理的重要文献，可反复阅读。

## 1. 理论基础

首先,PostgreSQL 实现 SSI, 同样也是基于 9.4.1 节所讲述的理论基础。在《Serializable Snapshot Isolation in PostgreSQL》这篇论文中, 在 9.4.1 节所讲述的理论基础上, 给出一个定理和一个推论, 原文如图 9-11 和图 9-12 所示。

**Theorem 1** (Fekete et al. [10]). *Every cycle in the serialization history graph contains a sequence of edges  $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$  where each edge is a rw-antidependency. Furthermore,  $T_3$  must be the first transaction in the cycle to commit.*

图 9-11 SSI 技术的定理

**Corollary 2.** *Transaction  $T_1$  is concurrent with  $T_2$ , and  $T_2$  is concurrent with  $T_3$ , because rw-antidependencies occur only between concurrent transactions.*

图 9-12 SSI 技术的推论

图 9-11 所述的定理, 说明在一个写偏序发生的有向图环中, 必然存在某三个事务  $T_1$ 、 $T_2$ 、 $T_3$  之间的两条边均是读写依赖构成的, 并且, 处于最右端被事务  $T_2$  所指向的事务  $T_3$  一定是在环中第一个提交的事务。如果这样的情况发生, 则写偏序异常出现, 意味着违反了可串行化。所以回滚三个事务中的一个, 则能确保不发生写偏序异常, 即确保了序列化。这一点, 和 9.3.1 节所述是一致的。

而读写依赖发生在并发的事务之间, 即写偏序发生一定是在并发的事务之间。所以 9.4.2 节所讲述的算法中, 对于读、写操作, 判断的都是正在执行的事务 (如判断条件是“rl.owner is running”, 而存在已提交的则回滚当前事务, 当前事务则是正在运行的事务), 而这些正在执行的事务之间要想形成“依赖”关系, 则被依赖的一定应该先提交才能形成正确的结果 (即写先进性才能被读操作读到所以才能形成读对写的依赖)。相对于  $T_1$  而言,  $T_2$  是写操作所以  $T_2$  应该先提交, 相对于  $T_2$  而言,  $T_3$  是写操作所以  $T_3$  应该先提交, 所以三个事务中最右端的  $T_3$  应该是第一个提交的。

图 9-12 所述的推论, 上面也进行了表述, 主要是说, 发生读写依赖的事务, 一定是并发的事务间, 才可能构成读写依赖。

其次, SSI 技术, 就是基于理论, 在并发的、尚在运行的事务之间做检测, 检测到图中包括图 9-13 的情况, 则回滚其中的一个事务<sup>⊖</sup>, 这样就能阻止环形成。

a sequence of edges  $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$

图 9-13

从并发的角度看, MVCC 技术相比 S2PL 和 OCC 允许而言可有更大的并发度 (参见

⊖ 挑选  $T_1$ 、 $T_2$ 、 $T_3$  中的哪个事务做回滚, 是一个需要仔细考虑的问题, 后续章节会谈及。



2.2.1 节“5. 锁的并发度”)，所以 MVCC 称为数据库引擎在事务管理方面的主流技术。而 MVCC 技术和 S2PL 技术结合，演变出诸如 MySQL 的事务处理实现方式，实则是做了一个折中。但是 SSI 技术，却是允许更大并发度的一个技术。为什么这么讲呢？

## 2. 并发度

SSI 技术是保证可串行化的技术。所以应该在可串行化这个层次去做比较。当 S2PL 技术在序列化的时候，读-写操作是不能同时发生的（当然是指在同一数据项上的并发读写操作）。但是 SSI 技术却是允许这样的事情“可能”发生。例如 SSI 技术中发生一个读写依赖 (rw-dependency) 或多个但不连续的读写依赖，是允许的，这样相比 S2PL 技术则有了更高的并发度，所以从并发的角度看，SSI 技术更优。

## 3. 只读事务

PostgreSQL 支持只读事务，在一个事务开始的时候，通过“BEGIN TRANSACTION READ ONLY”命令告知事务管理器，这是一个只读事务，不需要修改数据。

对于写偏序，当如图 1-2 所示的情况发生时，正是有一个事务是只读事务，此只读事务为图 1-2 中的事务 T1。在《Serializable Snapshot Isolation in PostgreSQL》这篇论文中，有如下定理，对此种情况做了讨论，对 SSI 技术做了进一步的优化。

**Theorem 3.** *Every serialization anomaly contains a dangerous structure  $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ , where if  $T_1$  is read-only,  $T_3$  must have committed before  $T_1$  took its snapshot.*

图 9-14 只读事务的定理

这个定理表明，对于图 9-14 中的三个事务，如果事务 T1 是只读的事务，那么事务 T3 一定是在事务 T1 获取快照之前就已经提交。

为什么这么说呢？根据第一个定理，如果发生写偏序，那么可以知道事务 T3 一定是在事务 T1 之前就已经提交。所以讨论事务 T1 是只读事务的情况。写偏序发生意味着有环存在，则事务 T1 之前的事务，假设为事务 T0，事务 T0 指向事务 T1 的边 ( $T0 \rightarrow T1$ ) 不可能是写读依赖和写写依赖，这是因为事务 T1 是只读的、且不会有写操作。因此事务 T0 指向事务 T1 的边只能是写读依赖，所以这意味着事务 T0 的写操作对于事务 T1 是可见的（注意现在讨论的是写偏序已经发生后的情况），因此 T0 一定是 T1 之前的已经提交的事务。根据之前的推论可知，事务 T3 应该是环中第一个提交的事务，因此 T3 应该在 T0 之前提交，所以 T3 在 T1 之前已经提交。一种特殊情况是，事务 T3 就是事务 T0。

这个定理有什么意义呢？这对于 SSI 技术中对写偏序问题的判断有价值。SSI 技术不检测环的存在，而是检测是否存在图 9-13 的情况，如果是，则挑选其中的一个事务进行回滚，以消除“可能”构成环的可能性。因为有第二个定理，可以对这个判断进行核实，看是否“误判”。

如果检测到图 9-13 的情况，因为 PostgreSQL 可以明确知道事务是只读的（另一种特例是检测是否一直没有发生过写操作），所以继续判断“如果事务 T0 是只读的，除非事务 T3 在 T1 获取快照之前就已经提交”这种情况是可以回滚事务 T1 或 T2（因为 T3 已经提交）的，否则就不进行回滚操作。这样就限制了回滚带来的“误杀”，提高了 SSI 技术的识别写偏序情况发生的正确判断率。

#### 4. Safe Snapshots

在图 9-13 中，如果处于中间的事务 T2 不会因存在读写依赖被 T1 指向，或者不会因存在读写依赖指向 T3，即 T2 不存在，则环也不可能构成。这样，即使 T1 是只读事务，也不需要检测其是否会构成 SSI 中的环，即不需要使用 SSI 中的检测技术。基于此点，论文中提出了“Safe Snapshots”的概念并在 PostgreSQL 中加以实现。论文中定义的“Safe Snapshots”概念如下：

Safe snapshots: A read-only transaction T has a safe snapshot if no concurrent read/write transaction has committed with a rw-antidependency out to a transaction that committed before T's snapshot, or has the possibility to do so.

这表明，并发的事务中，如果不存在 T2，则只读事务 T1 就会有一个安全快照（safe snapshot）。所以不能构成环，事务 T1 就不会被回滚，所以也不需要只在只读的事务 T1 上施加之前提到的“SIREAD”锁，这样就能精准地避免图 1-2 中描述的写偏序。

但是，实践中却没有办法提前识别出这种情况，即没有办法提前判断一个只读事务是否拥有一个“安全快照”，只有随着并发的事务执行，才能根据事务的执行情况择机判断。PostgreSQL 提供了一个并发事务列表，一个只读事务正常加“SIREAD”锁并维护 SSI 所需要的状态，直到一些事务完成提交，即在一些并发事务完成提交后，这时才能通过宏 SxactIsROSafe 判断事务 T1 的快照是否是安全的（判断关键是事务 T2 的不存在，这样就符合了安全快照的定义），这样就可以提前把事务 T1 上的“SIREAD”锁去掉，并且让事务 T1 在可重复读隔离级别下运行（使用 SI 技术），避免后续继续对事务 T1 使用 SSI 的技术做检测。这样能够提高只读事务 T1 后半段的执行效率。一个特例是，当并发的的事务中不存在写事务时，一个只读事务的快照可以被立刻认定是安全的。

#### 5. Deferrable Transactions

只读事务可以获得安全快照的一个特殊类型，称为“Deferrable Transactions”，其含义是某个只读事务，发起执行后，事务管理器暂不“立刻”执行，而是推迟一会，直到发现没有并发的写事务存在，这时，这个只读事务就可以获取一个安全快照，然后开始执行。这样的事务通过“BEGIN TRANSACTION READ ONLY, DEFERRABLE”告知事务管理器一个只读事务可推迟执行。

延迟只读事务执行，这是因为想获得一个安全快照，而安全快照的好处对于一个只读事务很有意义：

❑ 对于一个长事务，如使用 `pg_dump` 逻辑导出大批量的数据的时候，就需要一个安全快照，这样就不能对系统的写操作造成大的性能影响。

❑ 再如事务型系统，有时候也需要周期性地执行一些大规模的查询，这时，同理也需要一个安全快照以减少对系统事务型操作的影响。

推迟只读事务使其在能够获得安全快照时再开始执行，这样做的原因如下：

❑ 如果不能获得安全快照，则因需要进行 SSI 判断而在只读事务上施加“SIREAD”锁，这可能会导致大量的“SIREAD”锁被施加而消耗大量内存，而且一旦施加还不能被及时释放。而获得安全快照则如前所述可以避免施加“SIREAD”锁。

❑ 而且，非安全快照的只读事务更容易造成写偏序异常，因而需要参与到 SSI 技术的检测中耗费 CPU。

❑ 获得安全快照的事务，也不会面临被回滚的风险。

## 6. 锁管理器

SSI 技术主要是能够发现读写冲突（即前述的读写依赖）。所以需要对读操作进行检测和判断。

但是，在任何隔离级别下，PostgreSQL 都没有实现过读锁，这样就不能获取新的 SIREAD 模式的读锁（参见 9.3 节），所以也不可能标记出读写依赖。另外，PostgreSQL 的元组级写锁存储在元组头中，而不是内存表中，这导致难以有简单的方法用来检测读写冲突（每次判断是否存在读写冲突则遍历所有元组是不现实的事情）。

PostgreSQL 的 SSI 锁管理器仅存储 SIREAD 锁，不支持任何其他锁定模式，所以不会产生阻塞（读锁不阻塞任何操作）。而 SIREAD 锁的施加，是发生在读操作进行的时候：

❑ 如果是表扫描，则需要在读取元组的时候，为每个元组加 SIREAD 锁。

❑ 如果是索引扫描，则需要在索引页面上施加 SIREAD 锁。这样做的目的是为了节约内存空间。

❑ 上述两种方式，和 MySQL 的 InnoDB 有很大不同，首先 InnoDB 只在索引上加锁，但是，InnoDB 没有在页面级别上加锁，而是在索引的记录级别上加锁（索引记录上加间隙锁，参见 11.3 节），这样锁的粒度小但是锁表会很大。

PostgreSQL 的 SSI 锁管理器主要作用有两个：一是在是在关系（relation，表或视图对象）、页面（page）或元组（tuple）上获取 SIREAD 锁，二是在写入元组时检查 SIREAD 锁以判断是否存在冲突的读写依赖，检查是否存在 SIREAD 锁的过程，是一个锁的粒度从粗到细的过程，先检查 relation、之后是 page，最后才是 tuple，这样使得 PostgreSQL 提供了锁升级的功能<sup>①</sup>。

① PostgreSQL 提供锁升级的另外一个原因是在 DDL 操作发生的情况下，如 `CLUSTER` 和 `ALTER TABLE` 执行时，可能会造成物理元组的位置发生变化，此时这些元组上的 SIREAD 锁将不再合法，于是 PostgreSQL 把锁升级到 relation 级别，以保证逻辑的正确性。同样的，索引上也存在类似的问题。但是，MySQL 却不会发生这样的事情，因为 MySQL 采用的是 S2PL，读锁会抑制 DDL 操作发生。



另外，事务提交后，其产生的 SIREAD 锁需要得到保留，因此 SIREAD 锁表可能会很大（因此 PostgreSQL 采取了很多节约内存的措施）。

SIREAD 锁不存在冲突，所以不需要进行死锁检测。

## 7. 冲突检测与解决

PostgreSQL 的 SSI 实现，依赖于两部分内容：一是使用了其已有的 MVCC 的多版本数据，二是实现了一个新的锁管理器来检测读写冲突。使用哪一个部分检测判断，取决于写和读操作按时间发生的顺序：

- 如果先写：写操作发生，则可以从 MVCC 的多版本数据推断写冲突（写冲突导致有新版本产生），而不使用任何锁。
- 如果先读：读操作发生，则每当事务读取元组时，需要进行可见性判断（参见 9.2.1 节），通过检查元组的 xmin 和 xmax，能够确定元组是否在事务的快照中可见。
  - 如果元组不可见，则会存在读写冲突。这是因为创建它的事务在读取操作发生前生成的快照，在此快照下元组不可见表明此元组还没有提交，这就是读写冲突（生成快照的读在前，生成不可见版本的元组的写在后）。
  - 如果元组已被删除，则会存在读写冲突。元组已被删除即元组有一个 xmax 值，但是此元组仍然可以被读到，这就是读写冲突（生成快照的读在前，删除可见的版本的写在后）。
  - 如果先发生读，则还需要处理在写入之前发生读取的情况。这个时候，不能只使用 MVCC 数据，需要使用 SIREAD 锁跟踪读取的依赖关系。

在 9.4.2 节“理论基础”部分，我们得知，一个读写冲突是不足以为 SI 技术带来写偏序异常的。只有满足如图 9-13 所示的情况，才可能会发生写偏序，才需要 SSI 技术来解决写偏序异常。而 PostgreSQL 为每个事务保留所有读写依赖（rw-antidependencies）列表，但不保留写-写和写-读依赖关系，这个事务列表是按事务提交的顺序有序的。所以 PostgreSQL 的 SSI 技术，就是使用事务列表检测出两个读写依赖如连续相邻，则回滚其中一个事务以消除发生写偏序的隐患。

但是，回滚操作，存在着一些值得讨论的话题。第一个就是“Safe Retry”原则。论文中讲述如下：

Safe retry: if a transaction is aborted, immediately retrying the same transaction will not cause it to fail again with the same serialization failure;

这表明，满足写偏序异常的两种事务操作（图 1-1 和 1-2）。其中某个事务是可以被回滚而不用担心会给系统带来负面影响。“Safe Retry”原则要求某个事务被回滚后再执行，不会再次造成同样的写偏序异常。但是，这不表明发生写偏序异常的三个事务任何一个都可以被“随机”回滚（见图 9-13），其中已经提交的事务是不可以回滚的，只能从没有提交的事务中选择一个进行回滚，通常的倾向是选择三个事务中的第二个事务回滚（如图 9-13

中的 T2 事务), 但不尽然。总的而言, 回滚的规则如下 (如图 9-13 所示):

- ❑ 第一种情况: 不回滚 T1 或 T2 事务, 直到 T3 事务提交。
- ❑ 第二种情况: 回滚 T2 事务 (因为上一条, T3 事务已经提交)。这么做是因为 T3 已经提交, 重新执行 T2 事务不再会和 T3 事务同时并发执行, 所以重新执行 T2 事务则不会再次发生同样的读写冲突。如果回滚 T1 事务, 则 T1 事务再次执行时, 如果 T2 事务依然在执行, 则还可能会发生同样的如图 9-13 所示的情况。
- ❑ 第三种情况: 如果 T2、T3 事务都已经提交, 没得可选, 只有 T1 事务可被回滚。但是这符合 “Safe Retry”, 因为 T1 事务再次执行后不可能与已经提交的 T2、T3 事务处于并发状态。
- ❑ 另外, 对于第二种情况, 特例情况如 XA 支持的 2PC 阶段, 如果事务 T2 已经处于 “PREPARED” 状态, 则 T2 事务是必须提交的, 这时只能选择回滚 T1 事务, 尽管这可能不符合 “Safe Retry”。

## 8. 内存相关问题

PostgreSQL V9.1 为 SSI 实现了新的事务管理器, 此管理器需要记录读数据项的情况。但是一个事务提交后, 这样的信息还需要用于其他事务做读写依赖的判断, 即事务读记录的这个信息不能随着事务的结束而丢弃, 如写偏序中的第二种情况, T1 事务是只读的, 但其提交后还需要保留其 “历史上曾经读过数据项 X” 的信息, 用以为 T2 事务判断是否存在读写依赖提供依据。此种情况下, 假如 T2 事务是一个长事务, 则之前大量的类似 T1 事务所持有过的 SIREAD 锁的信息就被迫保存在事务管理器中, 耗费了大量的内存资源。一旦内存被耗费干净, 则整个数据库引擎将难以正常工作。

为解决这个问题, PostgreSQL 提供了四种特性 (摘自论文):

- ❑ Safe snapshots and deferrable transactions can reduce the impact of long-running read-only transactions
- ❑ Granularity promotion: multiple fine-grained locks can be combined into a single coarse-grained lock to reduce space.
- ❑ Aggressive cleanup of committed transactions: the parts of a transaction's state that are no longer needed after commit are removed immediately
- ❑ Summarization of committed transactions: if necessary, the state of multiple committed transactions can be consolidated into a more compact representation, at the cost of an increased false positive rate

前面的两种, 在之前已经讲述过, 接下来我们讲述后面的两种。

一个事务持有 SIREAD 锁的信息, 当与其并发的的事务都提交, 则持有 SIREAD 锁的信息可以被释放 (读写倚赖只能发生在并发事务之间)。因此, 当最老的活动事务提交时, 可以清理 SIREAD 锁的信息。这就是 “Aggressive cleanup of committed transactions”。所

以事务提交时，存在 SIREAD 锁清理的操作，这样的操作，由 ClearOldPredicateLocks() 函数完成，如图 9-15 所示。但是，事务提交的标志，不仅是事务提交的函数，而且还有诸如 SxactGlobalXminCount 这样的变量，当其值为零，则可以清理 SIREAD 锁。比如，GetSafeSnapshot() 函数会等待并发事务完成，这样一个只读事务将是安全的事务所以可以清理 SIREAD 锁。

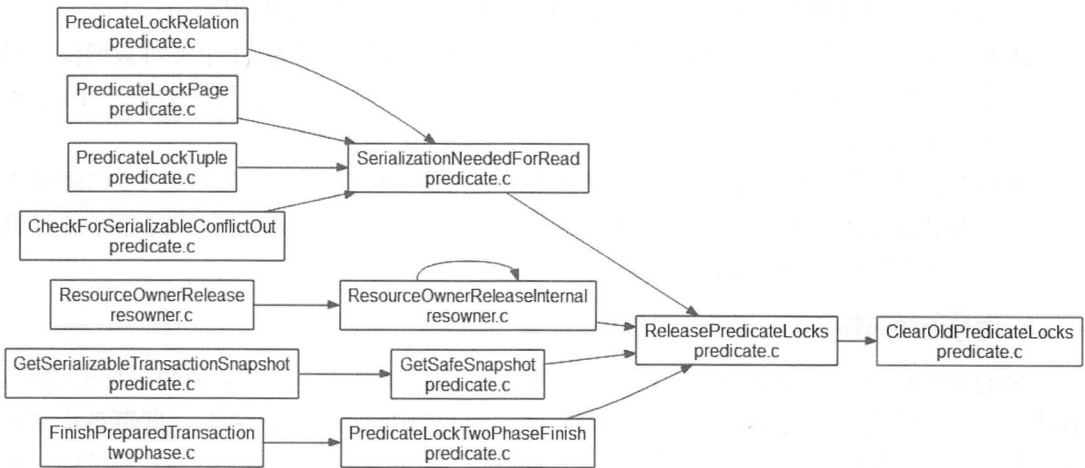


图 9-15 谓词锁释放操作调用关系上下文图

SSI 技术下的锁表容量有限（节约内存），当已经提交的事务有很多时却不能被释放，则锁表会用光。为了允许接入更多的新事务，PostgreSQL 提供了一种事务汇总技术，即把已经提交但不能被释放的事务的信息汇总在一个“dummy”事务上，然后释放那些事务，这样就能省出锁表空间而接纳新的事务。这样的技术被称为事务汇总。

那么，什么样情况下事务可以被汇总呢？

第一种情况：对于一个活动的事务，当其要修改一个元组，则需要检查这个元组上，是否曾经有已经提交了的事务发生过读操作？如果是，则如图 9-16 所示，可能构成一个危险的连续的两个读写依赖，为了检测是否存在这样的危险，则已经提交的事务上需要保留 SIREAD 锁信息，但是这个已经提交了但保留 SIREAD 锁信息的事务对于当前活动的事务而言，是不需要知道已经提交的这个事务是谁，而只需要知道有一个持有 SIREAD 锁的事务曾经在本事务要修改的元组上。因此，可以把已经提交的事务上的 SIREAD 锁等信息汇总到一个“dummy”事务上，然后释放这个已经提交的事务。

第二种情况：对于一个活动的事务，当其要读一个元组，则需要检查这个元组上是否存在并发事务的写操作。如果是，则存在两种子情况，分别如图 9-17 和 9-18 所示，这两种子情况可能导致危险的连续的两个读写依赖。这时，因为判断本读事务和其他写事务的关系，所以可以从元组头中找出写操作对应的事务信息，即进行元组可见性判断。另外，还需要检查写事务是否是序列化隔离级别的。更进一步，对于图 9-18 的情况，需要确认



第三个事务一定是已经提交的事务，这时，因为存在事务汇总的情况，作为已经提交了的事务被汇总到一个“dummy”事务上而从事务链上去除，所以不能再根据事务链进行事务相邻且存在读写倚赖的判断了，而是根据一个从汇总事务 ID 到标识读写倚赖关系出边（a conflict out）的最老事务的提交顺序序号的一个映射表进行判断（代码中注意 SlruCtlData 结构体、OldSerXidAdd()、OldSerXidGetMinConflictCommitSeqNo() 等函数的使用，本书不再多述）。

事务的汇总，通过 SummarizeOldestCommittedSxact() 函数调用 ReleaseOneSerializableXact() 函数完成。

$$T_{committed} \xrightarrow{rw} T_{active} \xrightarrow{rw} T_3$$

图 9-16 读写依赖（一）

$$T_1 \xrightarrow{rw} T_{active} \xrightarrow{rw} T_{committed}$$

图 9-17 读写依赖（二）

$$T_{active} \xrightarrow{rw} T_{committed} \xrightarrow{rw} T_3$$

图 9-18 读写依赖（三）

## 9. 小结

9.4 节以论文《Serializable Snapshot Isolation in PostgreSQL》为依据，讲述了 PostgreSQL 可串行化隔离级别的实现技术。而 9.3 节以论文《Generalized Isolation Level Definitions》为依据，讲述了 SSI 实现的基本原理和算法。请注意，在理解 PostgreSQL 的 SSI 技术时，既要注意两者相同之处，又要注意不要把这两篇论文混加在一起理解。

两者基于的原理是一样的，但实现方式不同。

给出论文《Generalized Isolation Level Definitions》的算法实现，是想帮助读者从宏观上理解实现 SSI 技术需要考虑的重要的点，因为其给出的算法非常的简洁明了，所以读者要理解算法背后描述的要素。

PostgreSQL 的实现以论文《Serializable Snapshot Isolation in PostgreSQL》为依据，该文较前篇论文讨论了更多更为深入的细节，基于这些细节，PostgreSQL 实现了相应的代码，并以不同的实现方式表达了同样的思想。诸如“SIREAD”锁的实现不同、表示读写冲突的方式不同、冲突检测的方式不完全相同等。掌握 PostgreSQL 的 SSI 技术，则务必要多读这篇论文<sup>①</sup>。

### 9.4.3 谓词锁数据结构

论文中提及的“SIREAD”锁，在 PostgreSQL 中被称为谓词锁。

谓词锁相关的结构体之间的关系可参见图 9-19，各个结构的具体解释可参见 9.5.1 节。

① 本篇论文中涉及了其他的一些重要话题，诸如 2PC、流复制技术、保存点、子事务等，并书不再提及，请参阅论文。

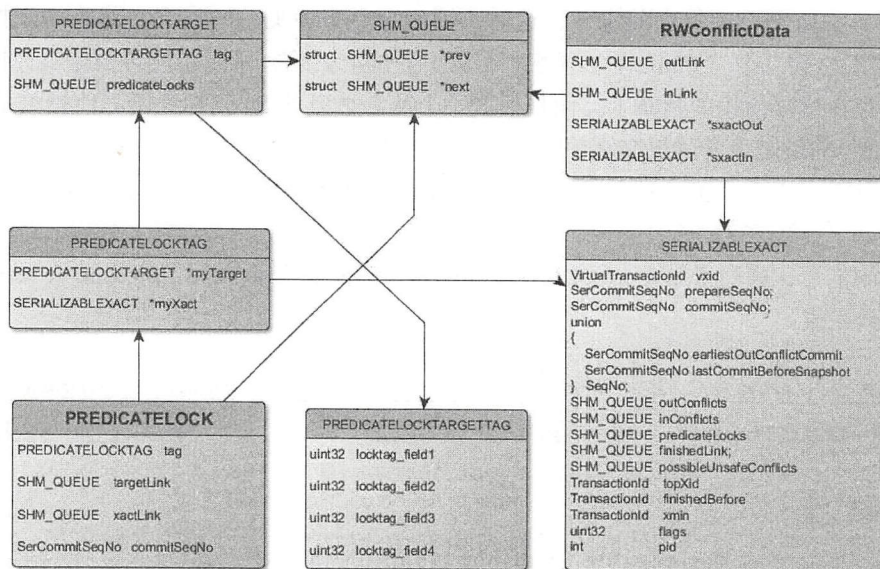


图 9-19 谓词锁相关数据结构图

### 1. SIReadLock (谓词锁)

论文中,把“以快照方式读取过此数据的某个版本”中的读操作称为 SIRead Lock,而 PostgreSQL 中则称之为“Predicate”锁,即谓词锁。主要代码位于 predicate.c、predicate.h、predicate\_internals.h 这几个文件中。

PostgreSQL 使用“PREDICATELOCKTARGET”表示一个谓词锁的对象,然后在此对象上施加“PREDICATELOCK”表示的谓词锁。

```
// The PREDICATELOCKTARGET struct represents a database object on which there are predicate locks.
typedef struct PREDICATELOCKTARGET //在一个“对象”上的谓词锁
{
    /* hash key, 一个标志, 唯一表示一个对象 */
    PREDICATELOCKTARGETTAG tag; /* unique identifier of lockable object */

    /*谓词锁对象队列表 */
    SHM_QUEUE predicateLocks; //“PREDICATELOCK”对象的队列表,其结构体如下
} PREDICATELOCKTARGET;
```

“PREDICATELOCK”对象的定义如下,一个谓词锁对象有一个唯一的使用“PREDICATELOCKTAG”定义的标识tag,然后包括两个列表(一个是表示存在谓词锁对象的“PREDICATELOCKTARGET”,一个是表示事务间冲突关系的“SERIALIZABLEXACT”):

```
typedef struct PREDICATELOCK //谓词锁
```



```

{
    PREDICATELOCKTAG tag;           //谓词锁的唯一的标识，被当作hash key以唯一标识谓词锁
    SHM_QUEUE targetLink;           /* list link in PREDICATELOCKTARGET's list of
                                     predicate locks */
    SHM_QUEUE xactLink;             /* list link in SERIALIZABLEXACT's list of
                                     predicate locks */
    SerCommitSeqNo commitSeqNo;     /* only used for summarized predicate locks */
    // “summarized predicate”，汇总事务，是为节约内存谓词锁表而采取把已经提交的多个事务
    的相关谓词汇集组合到一个“dummy”事务上
} PREDICATELOCK;

```

当一个谓词锁对象有一个唯一的使用“PREDICATELOCKTAG”定义的标识tag，此tag则由两个结构体构成：

```

typedef struct PREDICATELOCKTAG
//谓词锁的标识。把一个数据库对象、对象上的谓词锁和一个事务绑定在一起
{
    PREDICATELOCKTARGET *myTarget; //用一个四元组表示一个数据库对象
    SERIALIZABLEXACT *myXact;      //一个序列化的事务
} PREDICATELOCKTAG;

```

所以，PostgreSQL使用“PREDICATELOCK”结构体表示一个谓词锁，然后用一个数据库的对象作为此谓词锁的标识（tag）来标识一个谓词锁，这表明谓词锁是数据库对象和事务间的一个特定关系，只是这样的关系是用以表示事务的读操作的状态。

## 2. 谓词锁的类型和设置方式

PostgreSQL在加锁前，先设置锁的标识然后调用PredicateLockAcquire()函数完成加锁操作，谓词锁加锁如9.5.2节所示，本节分析谓词锁的可施加的对象，一共有三种类型，分别对应的是不同的粒度，最粗的粒度是关系（表、视图等对象），其次是物理页面，粒度最细的是元组，具体表示方式如下：

```

typedef enum PredicateLockTargetType
//谓词锁只在3种对象上加锁，也表明谓词锁的粒度是在3种对象上
{
    PREDLOCKTAG_RELATION, //关系级谓词锁
    PREDLOCKTAG_PAGE,     //物理页面级谓词锁
    PREDLOCKTAG_TUPLE     //元组级谓词锁
    /* TODO SSI: Other types may be needed for index locking */
} PredicateLockTargetType;

```

对于每一种谓词锁，设置此谓词加锁的方式如下：

```

#define SET_PREDICATELOCKTARGETTAG_RELATION(locktag,dboid,relid) \
//设置元组级谓词锁
    (locktag).locktag_field1 = (dboid), \
    (locktag).locktag_field2 = (relid), \

```



```

(locktag).locktag_field3 = InvalidBlockNu, \
(locktag).locktag_field4 = InvalidOffsetNumber)
#define SET_PREDICATELOCKTARGETTAG_PAGE(locktag,dboid,relid,blocknum) \
//设置物理页面级谓词锁
((locktag).locktag_field1 = (dboid), \
(locktag).locktag_field2 = (relid), \
(locktag).locktag_field3 = (blocknum), \
(locktag).locktag_field4 = InvalidOffsetNumber)
//页面级加锁,所以元组位置置InvalidOffsetNumber值
#define SET_PREDICATELOCKTARGETTAG_TUPLE(locktag,dboid,relid,blocknum,offnum) \
//设置元组级谓词锁
((locktag).locktag_field1 = (dboid), \
(locktag).locktag_field2 = (relid), \
(locktag).locktag_field3 = (blocknum), \
(locktag).locktag_field4 = (offnum))

```

在设置谓词锁的时候,就是给锁的标志对象赋值,例如对于元组谓词锁,设置此锁意味着对在表示的是哪个数据库的哪个关系的哪个页面的哪个元组进行读操作从而需要记载其 SIREAD 锁值。

而谓词锁自身的表达,是通过一个结构体实现的,其注册在全局的 PredicateLockTargetHash 中,如下:

```

typedef struct PREDICATELOCK
{
    /* hash key */
    PREDICATELOCKTAG tag;          /* unique identifier of lock */    //锁的标志
    /* data */
    SHM_QUEUE    targetLink;      /* list link in PREDICATELOCKTARGET's list of
    predicate locks */           //指向施加谓词锁的数据库中的对象
    SHM_QUEUE    xactLink;        /* list link in SERIALIZABLEXACT's list of
    predicate locks */           //指向事务列表
    SerCommitSeqNo commitSeqNo; /* only used for summarized predicate locks */
} PREDICATELOCK;

```

而谓词锁的典型用法如下,包括三个步骤:

```

heap_fetch(...) //当获取元组的时候
{...
    valid = HeapTupleSatisfiesVisibility(tuple, snapshot, buffer);
    //首先检查元组的可见性
    if (valid) //如果元组可见,则施加谓词锁
        PredicateLockTuple(relation, tuple, snapshot);
    CheckForSerializableConflictOut(valid, relation, tuple, buffer, snapshot);
    //第三检查元组的读写依赖
    ...)
}

```

### 3. 可串行化事务

在事务的隔离级别为 `SERIALIZABLE` 时, PostgreSQL 为支持 SSI 技术, 使用 “`SERIALIZABLEXACT`” 表示可串行化的事务, 并标识事务与读写冲突之间的关系。此结构体中带有大量判断读写冲突的信息, 即有大量和其他事务之间的关联关系 (两个相邻的读写冲突可能会造成写偏序异常, 所以需要判断本事务和其他事务之间的读写关系所以需要保留本事务与其他事务之间的关系), 需要特别注意。

```
typedef struct SERIALIZABLEXACT // SERIALIZABLEXACT是一个可串行化的事务
{
    VirtualTransactionId vxid; /* The executing process always has one of these. */
    // 虚拟的事务ID
    /*
     * We use two numbers to track the order that transactions commit. Before
     * commit, a transaction is marked as prepared, and prepareSeqNo is set.
     * Shortly after commit, it's marked as committed, and commitSeqNo is set.
     * This doesn't give a strict commit order, but these two values together
     * are good enough for us, as we can always err on the safe side and
     * assume that there's a conflict, if we can't be sure of the exact
     * ordering of two commits.
     */
    /*
     * Note that a transaction is marked as prepared for a short period during
     * commit processing, even if two-phase commit is not used. But with
     * two-phase commit, a transaction can stay in prepared state for some
     * time.
     */
    // 以下两个变量, 用以表明事务的提交顺序
    SerCommitSeqNo prepareSeqNo; // 被PreCommit_CheckForSerializationFailure()
    函数赋值, 表示本事务还没有提交
    SerCommitSeqNo commitSeqNo; // 被ReleasePredicateLocks() 函数赋值, 一旦被赋值,
    表示本事务已经提交

    /* these values are not both interesting at the same time */
    union
    {
        SerCommitSeqNo earliestOutConflictCommit; /* when committed with conflict out */
        SerCommitSeqNo lastCommitBeforeSnapshot; /* when not committed or no conflict out */
    } SeqNo;
    // outConflicts和inConflicts用以表示事务间的相互影响, inConflicts是本事务影响了其他事务,
    outConflicts是其他事务影响了本事务
    SHM_QUEUE outConflicts; // 与本事务发生过读写冲突的写操作所在的事务列表。这样的
    事务的数据, 本事务是不可以读的
    SHM_QUEUE inConflicts; // 与本事务发生过读写冲突的读操作所在的事务列表。这样的
    事务, 是不可以读本事务的数据的
    SHM_QUEUE predicateLocks; // 事务上的谓词锁列表, 即本事务施加的SIREAD锁
}
```



```

    SHM_QUEUE    finishedLink;    /* list link in FinishedSerializableTransactions
*/ // “FinishedSerializableTransactions” 是全局共享的事务列表，表示已经完成的事务，即已
提交的事务。特别注意，已经完成的事务，依旧可能会和正在执行的事务构成读写冲突（这是快照隔离技术的
缺陷，因此才诞生了可序列化的快照隔离、即SSI技术）

/*
 * for r/o transactions: list of concurrent r/w transactions that we could
 * potentially have conflicts with, and vice versa for r/w transactions
 */
    SHM_QUEUE    possibleUnsafeConflicts; //对于只读事务而言，存在可能的不安全的冲突
（读写事务带来的不安全的冲突）

    TransactionId topXid;          /* top level xid for the transaction, if one
exists; else invalid */
    TransactionId finishedBefore; //表明本事务在什么时候可以被释放。即本事务被提交的时候，
即将发生的事务不再与本事务是并发的，所以可以安全的在其完成时，释放之前已经完成的、且不与其并发
的本事务。这表明了在SSI技术中一个序列化事务的生命周期中的生命结束段。故此，在ReleasePredicateLocks()
函数中为本事务的finishedBefore赋值表示本事务结束时即将启动的事务是哪个（称这样的事务为“未来
关联事务”），在之后ClearOldPredicateLocks()函数被触发后，“未来关联事务”才可以清理它出生
之前提交的事务

    TransactionId xmin;            /* the transaction's snapshot xmin */
    uint32        flags;           /* OR'd combination of values defined below */
    int           pid;             /* 进程的ID
} SERIALIZABLEXACT;

```

#### 4. 系统级的谓词锁表和可串行化事务表

在数据库引擎一层，需要知道可串行化事务和谓词锁等信息，这些信息，存放在谓词锁表等相关的 Hash 结构中（位于共享内存的 HASH 表），主要如下：

```

static HTAB *SerializableXidHash;
//所有序列化事务的ID注册到Hash表，通过此hash，可以找到所有的序列化事务
static HTAB *PredicateLockTargetHash;
//存放带有谓词锁的数据库对象，Hash表中的每个元素都是一个带有谓词锁的数据库对象
static HTAB *PredicateLockHash;
//把谓词锁和事务关联，Hash表中的每个元素都是一个带有谓词锁的数据库对象和其所在事务的信息
static SHM_QUEUE *FinishedSerializableTransactions;
//已经提交的序列化事务，在事务提交时把提交的事务注册到这个双向列表中

```

从 CreatePredicateLock() 等函数可以看出，这些 Hash 表是全局变量，对谓词锁表的查找等操作都依赖这些全局变量。

除了全局的谓词锁等对象外，还有本地谓词锁表，其作用是在一个会话进程内标识本进程内部的谓词锁，查找谓词锁时，优先查找本地的，如果不存在，才去全局范围内查找。如 PredicateLockAcquire() 函数先在 LocalPredicateLockHash 范围内查找。

```

static HTAB *LocalPredicateLockHash = NULL; //会话进程本地的谓词锁表

```



## 5. 全局的谓词事务表

PostgreSQL 为 SSI 技术实现了一个全局的谓词事务表,称为“PredXact”,用以管理整个引擎级别的全局可串行化事务的整体工作,所以“PredXact”是把握 SSI 技术的一个主要入口点。从形式定义上看,这是一个全局表,所以特别注意在调用 ShmemInitStruct() 做初始化时,其长度是固定的,即内存的使用是有限的(原因参见 9.4.2 节“内存相关问题”),换句话说,并发的可串行化事务的个数是有限的,PostgreSQL 设定每个会话平均 10 个可串行化事务。因此,这样的限制也影响了与其对应的全局可串行化事务 ID 表“SerializableXidHash”的大小。

```
/*
 * This provides a list of objects in order to track transactions participating
 * in predicate locking.
 * Entries in the list are fixed size, and reside in shared memory. The memory
 * address of an entry must remain
 * fixed during its lifetime. The list will be protected from concurrent update
 * externally;
 * no provision is made in this code to manage that. The
 * number of entries in the list, and the size allowed for each entry is fixed
 * upon creation.
 */
static PredXactList PredXact;
```

这个表是一个“PredXactListData”结构体,存放了每个可串行化事务的信息以及这个事务和其他可串行化事务之间的关系,序列化事务之间的关系用以判断读写依赖。

```
typedef struct PredXactListData //全局可串行化事务信息,内存容量固定,资源有限
{
    SHM_QUEUE    availableList; //可用的可串行化事务,即还没有被使用的可串行化事务“槽”。
    //每个新的可串行化事务创建,就调用GetSerializableTransactionSnapshotInt()获取可串行化快照
    //一次,即从availableList中拿出一个事务,“可用的”即减少一个。如果用光,即分配不成功,则需要调
    //用SummarizeOldestCommittedSxact()把已经提交的可串行化事务汇总,以释放出新“槽”点供分配。
    SHM_QUEUE    activeList; //活动状态的可串行化事务,每分配成功一个可串行化事务,即注册
    //到这个活动的可串行化事务列表中。当调用ReleaseOneSerializableXact()释放一个可串行化事务的时
    //候,才从活动的可串行化事务列表activeList中去除,归还给availableList。注意此列表是按照事务号
    //从小到大有序的,借以表明了事务之间的顺序

    /* These global variables are maintained when registering and cleaning up
    * serializable transactions.
    * They must be global across all backends, but are not needed outside the
    * predicate.c source file.
    * Protected by SerializableXactHashLock. */
    TransactionId SxactGlobalXmin; //所有的处于活动状态的可串行化事务中,快照建立
    //时间最早的(最老的活动的)事务,此事务之前的已经提交的事务的谓词锁等信息则可以被清理掉了。这意味
    //着清理那些与SxactGlobalXmin值对应的可串行化事务未曾并发执行过的、且已经提交的老事务
```

```

    int                SxactGlobalXminCount;    //有多少个活动的可串行化事务与最老的这个活动的
事务是并发的。一旦此值递减为零，则需要调用SetNewSxactGlobalXmin()函数找出下一个处于活动状态
的（没有提交也没有回滚也不是已经提交了的事务，重新统计在所有的活动事务中，有相同快照起点（xmin
值相同）的事务个数给此变量赋值

    int                WritableSxactCount;      //有多少个活动的发生写操作的序列化事务
    SerCommitSeqNo     LastSxactCommitSeqNo;    //最近的一个可串行化事务的提交号。提交了的可串
行化事务相对提交顺序靠这个变量排定（注意只是相对提交顺序，并发的任务之间写偏序的判断要依赖他们之
间的相对提交次序）

    /* Protected by SerializableXactHashLock. */
    SerCommitSeqNo     CanPartialClearThrough;  //可以从哪个已经提交的事务号开始清理谓词
锁和读写冲突中的读操作的事务。参见上一小节对“finishedBefore”的解释。注意，如果写事务个数
WritableSxactCount值为零，则可以大大加快已经提交的事务的清理工作（使用
LastSxactCommitSeqNo给CanPartialClearThrough赋值）

    /* Protected by SerializableFinishedListLock. */
    SerCommitSeqNo     HavePartialClearedThrough; /* have cleared through this seq no */
    SERIALIZABLEXACT *OldCommittedSxact;        //执行“汇总事务”，即把已经提交的事务的
一些信息汇集在一个dummy事务对象结构内
    PredXactListElement element; //可串行化的事务（实际上是可串行化的事务链）
}PredXactListData;

```

## 6. 读写冲突

在 9.3 节用 inConflict 和 outConflict 表示事务间的关系，这样的关系是两个事务之间的一条“边”，读写冲突导致事务有出边和入边，其中边依赖于事务对象（事务节点的两条边，一点连两线），判断冲突时是对会话线程这样的对象进行判断的。

但是 PostgreSQL 使用“RWConflictData”表示事务之间的读写冲突关系，但是却没有使用如上的方式，而是以“边”为主，用边连接起两个事务（一个边的两个顶点，一线连两点）。所以用 sxactOut 指向发出写操作的事务，用 sxactIn 表示发生读操作的事务，这样把两个事务放在一个结构体内，表示了一个读写冲突。这一点，可以很清晰地从 SetRWConflict() 函数中分析获得。

```

typedef struct RWConflictData    //事务之间，表示：“读写”冲突。是两个事务之间因“读写冲
突”建立起的联系
{
    SHM_QUEUE     outLink;        //自一个“sxact”发出的冲突链接
    SHM_QUEUE     inLink;        //指向一个“sxact”的冲突链接
    SERIALIZABLEXACT *sxactOut;  //SERIALIZABLEXACT是一个可序列化的事务，sxactOut表示
发生读操作的事务。有了指出的箭头方向
    SERIALIZABLEXACT *sxactIn;   //SERIALIZABLEXACT是一个可序列化的事务，sxactIn表示
发生写操作的事务。有箭头指向写事务之间
}RWConflictData;

```

即读写关系为：



读操作 -> 写操作 == **sxactOut** -> **sxactIn**

在 PostgreSQL 中，所有的读写冲突最后汇聚在“RWConflictPool”读写冲突池中。

```
typedef struct RWConflictData *RWConflict;

typedef struct RWConflictPoolHeaderData
{
    SHM_QUEUE    availableList;
    RWConflict    element;
}RWConflictPoolHeaderData; -----
                                   ↓
typedef struct RWConflictPoolHeaderData    *RWConflictPoolHeader;

static RWConflictPoolHeader    RWConflictPool; //读写冲突池，一个session中全局共享
```

而读写冲突池的操作关系如图 9-20 所示，从图中可以看出：

- 获取逻辑元组的时候（如调用 heapgettup() 函数），通过 CheckForSerializableConflictOut() 函数来检测和设置读写冲突。
- 在隔离级别为可串行化、通过 GetTransactionSnapshot() 函数获取事务的快照的时候，调用 SetPossibleUnsafeConflict() 函数设置事务间读写冲突的关系。

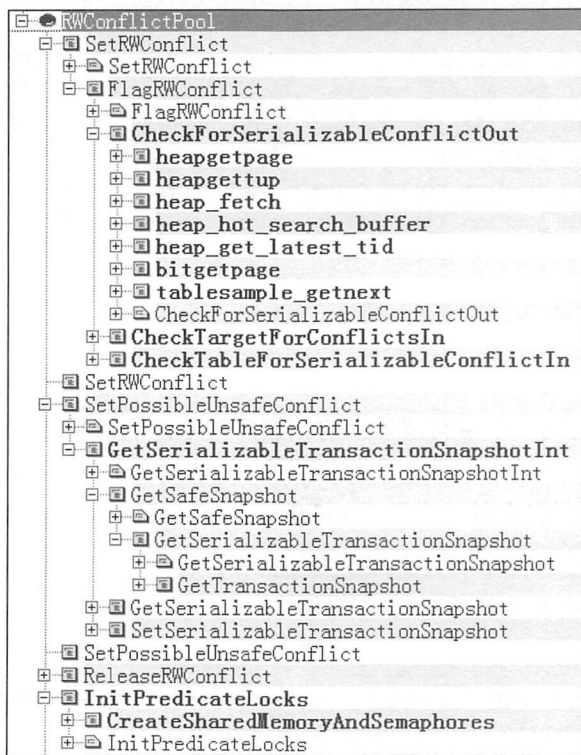


图 9-20 读写冲突池的操作关系



注意, `CheckForSerializableConflictOut()` 函数有如下一段注释:

```
* CheckForSerializableConflictOut
*   We are reading a tuple which has been modified.  If it is visible to
//第一种读写依赖, 即读写冲突
*   us but has been deleted, that indicates a rw-conflict out.
*   If it's not visible and was created by a concurrent (overlapping)
//第二种读写依赖, 即读写冲突
*   serializable transaction, that is also a rw-conflict out,
* We will determine the top level xid of the writing transaction with which
* we may be in conflict, and check for overlap with our own transaction.
* If the transactions overlap (i.e., they cannot see each other's writes),
* then we have a conflict out.
```

## 7. 冲突标志

PostgreSQL 定义了一些有用的标志, 表明特定的含义。

```
/* The following flag actually means that the flagged transaction has a
* conflict out *to a transaction which committed ahead of it*.  It's hard
* to get that into a name of a reasonable length.
*/
#define SXACT_FLAG_CONFLICT_OUT          0x00000010    //其他并发事务因读写依赖关系中、
                                                    发出写操作的事务
#define SXACT_FLAG_READ_ONLY            0x00000020    //只读事务
#define SXACT_FLAG_DEFERRABLE_WAITING   0x00000040    //只读且可延迟的事务
#define SXACT_FLAG_RO_SAFE              0x00000080    //只读事务, 拥有安全快照
#define SXACT_FLAG_RO_UNSAFE           0x00000100    //只读事务, 不拥有安全快照
#define SXACT_FLAG_SUMMARY_CONFLICT_IN  0x00000200    //汇总冲突的读事务
#define SXACT_FLAG_SUMMARY_CONFLICT_OUT 0x00000400    //汇总冲突的写事务
```

### 9.4.4 谓词锁操作

#### 1. 谓词锁创建

谓词锁的创建就是把一个带有谓词锁的数据库对象如表对象注册到谓词锁表中。在可串行化的隔离级别下, 如果一个事务需要获得关系(表对象)、页面或元组的时候, 就会在相应的对象上加锁(加锁的条件, 参见对 `PredicateLockTuple()` 函数的例举内容)。

```
CreatePredicateLock(const PREDICATELOCKTARGETTAG *targettag, uint32
targettaghash, SERIALIZABLEXACT *sxn)
{...
    /* Make sure that the target is represented. */
    target = (PREDICATELOCKTARGET *) //在“PredicateLockTargetHash”中增加(HASH链
表使用了HASH_ENTER_NULL)一个新的“PREDICATELOCKTARGET”对象, 这是一个被加锁的对象, 这个对
象可能是表、页面或一个元组
```



```

hash_search_with_hash_value(PredicateLockTargetHash, targettag,
                             targettaghash, HASH_ENTER_NULL, &found);
...
/* We've got the snext and target, make sure they're joined. */
locktag.myTarget = target; //把新增的“target”对象（代表数据库中的一个对象）保存到一个tag中，然后增加到“PredicateLockHash”里
locktag.myXact = snext;
lock = (PREDICATELOCK *) //为target对象创建一个锁对象，这个锁对象就是一个“谓词锁”，
所以，谓词锁就是“在指定的对象上施加了锁”从而保护了被加锁的数据库对象（表、页面或一个元组）
hash_search_with_hash_value(PredicateLockHash, &locktag, //(带有谓词锁的数据库对象被)增加到“PredicateLockHash”里
                             PredicateLockHashCodeFromTargetHashCode(&locktag, targettaghash),
                             HASH_ENTER_NULL, &found);
...
}

```

从图 9-21 中可以看出，创建谓词锁有两种情况：

- ❑ 当获取关系级、页级、元组级等上的谓词锁时，调用 PredicateLockAcquire() 完成谓词锁的创建，这种情况是数据库系统在运行期间施加谓词锁的主要的方式。
- ❑ 当系统做两阶段提交操作的恢复时，可以恢复（重新创建）谓词锁。

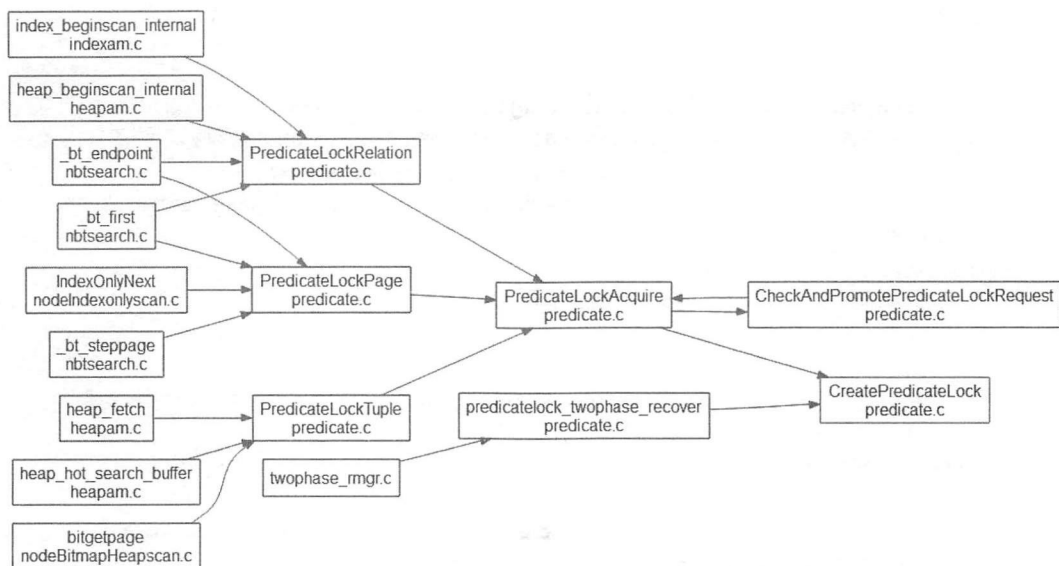


图 9-21 创建谓词锁表对象

## 2. 谓词锁加锁

谓词锁的施加，可能会伴随着锁从细粒度向粗粒度转变，而转变是有条件的。把细粒度的谓词锁转为粗粒度的谓词锁，目的是减小内存中锁表的大小。





### 1) 谓词锁加锁操作

谓词锁的加锁操作，和其他类型的锁不同，如 RegularLock 锁都是在特定的对象上加不同的标识，用以标志是否有锁存在，所以加锁的过程就是设置标志的过程。而谓词锁则不是设置特定的标志，其加锁过程是创建一个事务和其他事务的关系（CreatePredicateLock() 函数），即读写依赖关系，这样的关系，是表明本事务和其他事务之间是否存在若冲突（rw 冲突，详情参见原理部分）。

谓词锁的加锁操作，除了有加锁的功能外，还有锁升级的功能，即把细粒度锁向粗粒度锁升级（如元组级的锁向页级锁升级，这样的锁升级对于粒度变粗这种情况，对系统的性能没有影响，因为 RIREAD 锁不阻塞任何操作）；但没有提供锁自动降级的功能（如页级的锁向元组级锁降级，对于 RIREAD 锁没有必要降级）。

```
static void
PredicateLockAcquire(const PREDICATELOCKTARGETTAG *targettag)
//targettag表示准备要施加锁的对象；实现锁的加锁功能
{...
    if (CoarserLockCovers(targettag)) //如果要施加的锁被更粗粒度的锁包含，则不必再加锁，
        这是谓词锁优化的方式之一
        return;
    ...
    /* Acquire lock in local table */
    locallock = (LOCALPREDICATELOCK *)
        hash_search_with_hash_value(LocalPredicateLockHash⊖,
        //从会话里的本地HASH表里LocalPredicateLockHash找出细粒度锁的对象
        targettag, targettaghash,
        HASH_ENTER, &found); // "HASH_ENTER"
    表示要被查询的对象不存在则创建一个新对象，然后返回此对象
    locallock->held = true;
    if (!found) //没有找到则意味着这应该是一个新的锁，未被授予过
        locallock->childLocks = 0;

    //创建谓词锁，存放在全局“PredicateLockTargetHash”对象中，即全局谓词锁的锁表就是
    “PredicateLockTargetHash”
    CreatePredicateLock(targettag, targettaghash, MySerializableXact);

    /* Lock has been acquired. Check whether it should be promoted to a
     * coarser granularity, or whether there are finer-granularity locks to clean up. */
    if (CheckAndPromotePredicateLockRequest(targettag)) //如果锁的粒度太细，则把细粒
        度的锁升级为粗粒度的锁。目的是 减小内存谓词锁表的大小
    {...}
    else
    {
```

⊖ LocalPredicateLockHash: The local hash table used to determine when to combine multiple fine-grained locks into a single coarser-grained lock.





```

/* Clean up any finer-granularity locks */
if (GET_PREDICATELOCKTARGETTAG_TYPE(*targettag) != PREDLOCKTAG_TUPLE)
    //不是元组级的谓词锁，即是粗粒度的锁
    DeleteChildTargetLocks(targettag); //清理细粒度的谓词锁，能被清理的原因是锁
    升级过 (be promoted to a coarser-granularity lock)，细粒度的锁已经不再需要
}
}

```

## 2) 元组级谓词锁加锁逻辑, PredicateLockTuple()

元组级谓词锁是把一个元组的位置记载到一个标签（区别于其他类的谓词锁对象，在于元组在一个页面上有具体的偏移量）上，然后调用 PredicateLockAcquire() 函数获取一个谓词锁。其实现很简单，内容如下：

```

void
PredicateLockTuple(Relation relation, HeapTuple tuple, Snapshot snapshot)
{
    ...
    SET_PREDICATELOCKTARGETTAG_RELATION(tag, //设置关系级谓词锁的标志
                                         relation->rd_node.dbNode, relation->rd_id);
    if (PredicateLockExists(&tag)) //如果关系级谓词锁存在，则不施加元组级谓词锁。即粗粒
    度的谓词锁存在则不再授予细粒度的谓词锁
        return;

    tid = &(tuple->t_self);
    SET_PREDICATELOCKTARGETTAG_TUPLE(tag, //设置元组级谓词锁的标志
                                     relation->rd_node.dbNode, relation->rd_id,
                                     ItemPointerGetBlockNumber(tid), ItemPointerGetOf
                                     fsetNumber(tid)); //块号，块中的偏移
    PredicateLockAcquire(&tag); //施加元组级谓词锁
}

```

元组谓词锁的上下文调用关系，可参见图 9-22。

从图 9-22 可以看出，元组级谓词锁加锁有三种情况：

- 索引扫描直接定位元组：通过索引，可以直接定位元组，因此可以直接在元组上加谓词锁。
- 位图扫描直接定位元组：通过表示页面上元组分布（可用）的位图，可以直接定位元组，因此可以直接在元组上谓词加锁。
- TID 扫描直接定位元组：通过 TID（即用于表示元组物理位置的 t\_ctid），可以直接定位元组，因此可以直接在元组上加谓词锁，包括在元组上做 UPDATE/DELETE、在页面上做 INSERT、直接执行 TID 扫描方式获取元组等操作。

## 3) 页级谓词锁加锁逻辑, PredicateLockPage()

页级谓词锁是把一个页的位置记载到一个标签（区别于其他类的谓词锁对象，在于页面有一个页面号）上，然后调用 PredicateLockAcquire() 函数获取一个谓词锁。其实现很简单，代码不再罗列。



页级谓词锁的上下文调用关系，如图 9-23 所示。

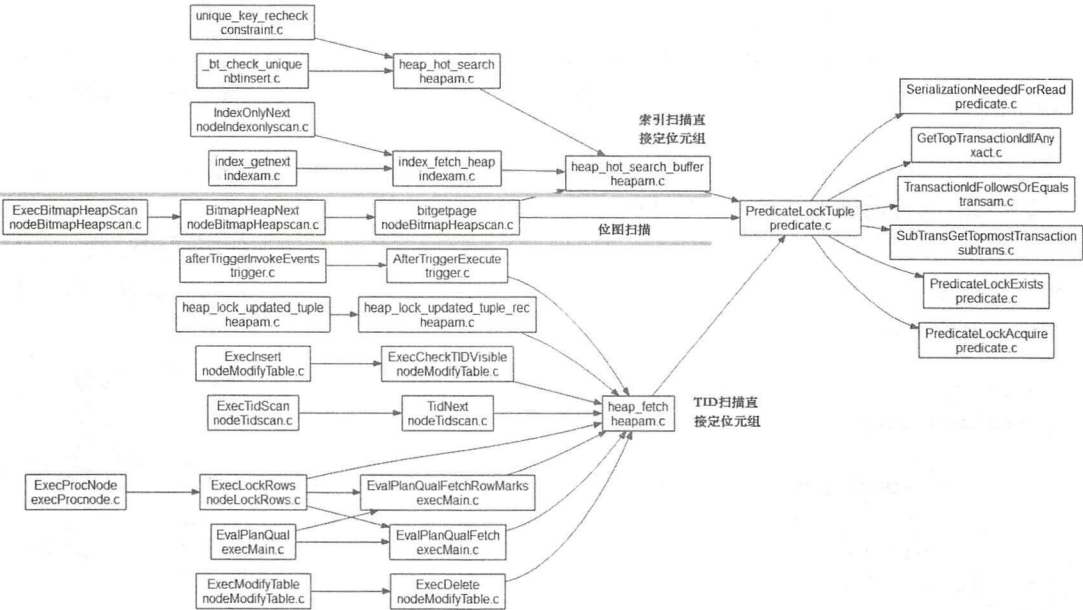


图 9-22 元组级谓词锁加锁上下文调用关系图

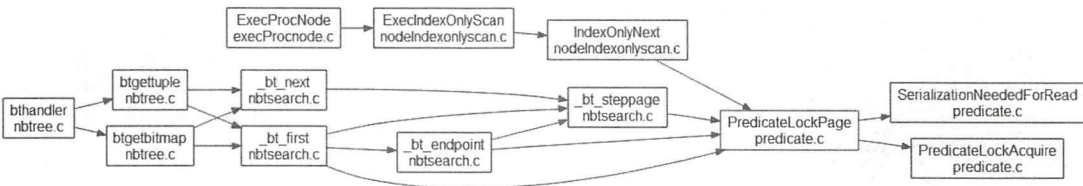


图 9-23 元组级谓词锁加锁上下文调用关系图

从图 9-23 可以看出，页级谓词锁加锁有两种情况：

- ❑ 索引扫描直接定位页面 (IndexOnlyNext() 函数)：通过索引（这里是“只索引扫描”。即“只从索引页面上获取数据”），可以直接定位页面，因此可以直接在页面上加谓词锁，此要求不用获取元组，直接操作的是索引的非叶子节点。
- ❑ B 树索引扫描直接定位页组 (\_bt\_first() 等系列函数)：B+ 树的扫描，是主要的索引方式。
- ❑ 通过以上两种方式，可以知道 PostgreSQL 在页面级施加的谓词锁，都是经索引施加的。所以在索引上施加谓词锁，粒度是页面，这点比 InnoDB 在索引的记录上施加记录锁的粒度相差较大，PostgreSQL 显得较为粗糙。

4) 关系级谓词锁加锁逻辑，PredicateLockRelation()

关系级谓词锁是把一个表的对象记载到一个标签（区别于其他类的谓词锁对象，在于表



对象有一个对象 ID 称为 `rd_id`) 上, 然后调用 `PredicateLockAcquire()` 函数获取一个谓词锁。其实现很简单, 代码不再罗列。

关系级谓词锁的上下文调用关系, 如图 9-24 所示。

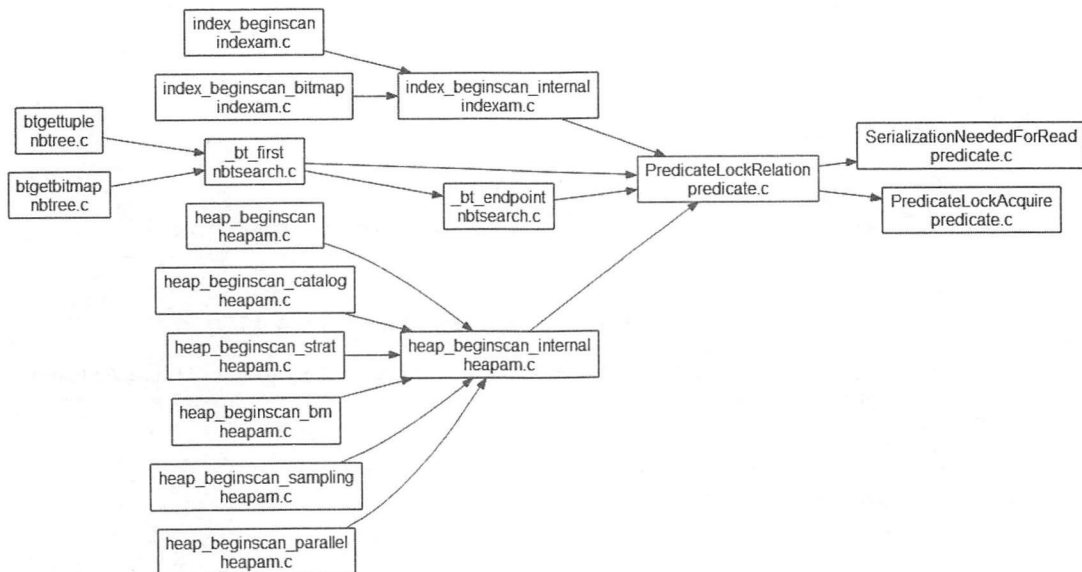


图 9-24 元组级谓词锁加锁上下文调用关系图

从图 9-24 可以看出, 关系级谓词锁加锁有两种情况:

- ❑ 索引扫描在关系上加关系级谓词锁: 首次扫描 (`index_beginscan_internal()` 函数、`_bt_first()` 函数), 在关系上施加谓词锁; 或者当索引为空 (`_bt_endpoint()`) 时, 直接在关系上施加谓词锁。
- ❑ heap 扫描在关系上加关系级谓词锁 (`heap_beginscan_internal()` 系列函数): heap 上的扫描 (如果不能使用索引、位图扫描, 则使用 heap 扫描完成顺序扫描的任务, 这意味着范围查找会在关系 (即表对象) 上加谓词锁以实现序列化的事务 (a serializable transaction))。

### 3. 谓词锁升级

细粒度的锁升级为粗粒度的锁, 根本目的是通过减少锁的个数减小锁表大小节约内存。而锁升级的条件其实很简单, 为每种锁升级设定一个阈值, 超过这个阈值则升级。如页级谓词锁多于 3 个则可以升级为表级谓词锁。

```

static bool
CheckAndPromotePredicateLockRequest(const PREDICATELOCKTARGETTAG *reqtag)
{...
    while (GetParentPredicateLockTag(&targettag, &nexttag))

```





```

//从孩子对象获得其父对象，例如元组所在的页是其父对象，页隶属于哪个表则表是页的父对象
{
    targettag = nexttag; //nexttag是获取到的父对象
    parentlock = (LOCALPREDICATELOCK *) hash_search(LocalPredicateLockHash,
    //先从本地谓词锁表里找
    &targettag, HASH_ENTER,
    //找不到则注册到本地谓词锁表
    &found);

    if (!found)
    {
        parentlock->held = false;
        parentlock->childLocks = 1; //为父对象拥有的孩子个数第一次计数
    }
    else
        parentlock->childLocks++; //为父对象拥有的孩子个数计数

    if (parentlock->childLocks >= PredicateLockPromotionThreshold(&targettag))
        //以个数为升级条件
    {
        promotiontag = targettag;
        promote = true;
    }
}

if (promote) //如果升级，则获取一个新的粗粒度的锁
{
    /* acquire coarsest ancestor eligible for promotion */
    PredicateLockAcquire(&promotiontag); //在新的粗粒度上获取谓词锁
    return true;
}
else
    return false;
}

```

#### 4. 谓词锁的释放

谓词锁的释放操作，和其他类型的锁不同，如 RegularLock 锁都是在特定的对象上取消特定的标识，所以释放锁的过程就是撤销加锁时设置的标志的过程，而且这类锁是随着事务的结束而被释放的。但是，谓词锁的释放锁过程是删除一个事务和其他事务的关系（ReleaseRWConflict() 函数）的过程，其释放的时机不是本事务结束而释放，而是滞后一段时间，需要等到所有与其并发执行的事务都结束才能真正释放。

“ReleasePredicateLocks()”函数用以完成谓词锁的释放（设置被释放的条件，而不是真正被释放），此函数比普通的 LWLock、RegularLock 锁的释放复杂得多，详情如下：

```

void
ReleasePredicateLocks(bool isCommit) //参数"isCommit"表明是提交或回滚；此函数用于释放谓词锁

```





```

{...
    //设置本次要被释放的事务将被清理的条件---当所有的与“本事务的并发事务”完成后被清理
    MySerializableXact->finishedBefore = ShmemVariableCache->nextXid;

    if (isCommit) //提交操作, 设置一些提交操作相关的标志
    {
        MySerializableXact->flags |= SXACT_FLAG_COMMITTED; /* 设置“已经提交”标志 */
        MySerializableXact->commitSeqNo = ++(PredXact->LastSxactCommitSeqNo);
        //并发的事务内的相对提交顺序
        /* Recognize implicit read-only transaction (commit without write). */
        if (!MyXactDidWrite)
            MySerializableXact->flags |= SXACT_FLAG_READ_ONLY;
        //如果不是写事务则标识为只读
    }
    else //回滚操作, 设置一些回滚操作相关的标志
    {
        MySerializableXact->flags |= SXACT_FLAG_DOOMED; /* 设置“将要回滚”标志 */
        MySerializableXact->flags |= SXACT_FLAG_ROLLED_BACK; /* 设置“已经回滚”标志 */
    }
}

if (!topLevelIsDeclaredReadOnly) //对于非只读的事务, 设置一个清理的范围, 留待将来被清理
{
    Assert(PredXact->WritableSxactCount > 0); //“PredXact”: 谓词事务列表, 参见
    9.4.3节“全局的谓词事务表”
    if (--(PredXact->WritableSxactCount) == 0) //如果写操作的事务数目为零, 则能够
    被清理的谓词锁的被清理的范围使用“CanPartialClearThrough”表示到当前最新的事务上
    {
        PredXact->CanPartialClearThrough = PredXact->LastSxactCommitSeqNo;
        // ClearOldPredicateLocks() 函数, 实现真正的谓词锁清理功能
    }
}
else //只读事务, 清理“可能”造成读写冲突的事务间的依赖“边”, 这样的“边”把两个事务“点”
连接起来, 清理就是去掉这样的“边”
{...
    possibleUnsafeConflict = (RWConflict) //谓词锁清理的是一种“可能”造成冲突的事务
    间的关系, 但不是说这样的关系一定能够造成冲突
    SHMQueueNext (&MySerializableXact->possibleUnsafeConflicts,
                  &MySerializableXact->possibleUnsafeConflicts,
                  offsetof(RWConflictData, inLink));
    while (possibleUnsafeConflict)
    {
        nextConflict = (RWConflict)
        SHMQueueNext (&MySerializableXact->possibleUnsafeConflicts,

```





```

        //不断找出下一个“只读事务”的读操作链，然后清理&possibleUnsafeConflict->inLink,
        offsetof(RWConflictData, inLink));

...

    ReleaseRWConflict(possibleUnsafeConflict); //释放“可能”的冲突。为什么此
    处能够释放掉可能的冲突呢？这是因为：第一，本事务，是只读事务；第二，指向本读事务的
    (inLink)，不可能构成读写依赖。
    possibleUnsafeConflict = nextConflict; //为刚找出的下一个做清理的准备
}

}

/* Check for conflict out to old committed transactions. */
if (isCommit //如果本事务是写事务（非只读），且有汇总的事务指向本事务，则存在读写依赖，
    本事务是被箭头指向的事务
    && !SxactIsReadOnly(MySerializableXact)
    && SxactHasSummaryConflictOut(MySerializableXact)) //有汇总的事务指向本事务
{
    /*
     * we don't know which old committed transaction we conflicted with,
     * so be conservative and use FirstNormalSerCommitSeqNo here
     */
    MySerializableXact->SeqNo.earliestOutConflictCommit = FirstNormalSerCommitSeqNo;
    MySerializableXact->flags |= SXACT_FLAG_CONFLICT_OUT; //标识本事务存在读写依
    赖，且是执行写操作的那一方
}

//Release all outConflicts to committed transactions. If we're rolling back
clear them all.
//Set SXACT_FLAG_CONFLICT_OUT if any point to previously committed transactions.
conflict = (RWConflict) SHMQueueNext(&MySerializableXact->outConflicts,
    &MySerializableXact->outConflicts,
    offsetof(RWConflictData, outLink));
while (conflict)
{
    //outLink,表明从本事务指出的冲突关系中遍历所有写事务，这些写事务写过的数据，本事务不能读取
    nextConflict = (RWConflict) SHMQueueNext(&MySerializableXact->outConflicts,
        &conflict->outLink, offsetof(RWConflictData, outLink));

    if (isCommit //（本事务）提交
        && !SxactIsReadOnly(MySerializableXact) //且不是只读事务
        && SxactIsCommitted(conflict->sxactIn)) //在冲突关系链中写操作的事务已经提交
    {
        if ((MySerializableXact->flags & SXACT_FLAG_CONFLICT_OUT) == 0
            || conflict->sxactIn->prepareSeqNo < MySerializableXact->SeqNo.
            earliestOutConflictCommit)
            MySerializableXact->SeqNo.earliestOutConflictCommit = conflict-
            >sxactIn->prepareSeqNo;
        MySerializableXact->flags |= SXACT_FLAG_CONFLICT_OUT;
        //标识本事务存在读写依赖，且是读操作的那一方
    }
}

```





```

    }

    if (!isCommit //不提交,但未必是因回滚操作引发。只是表明需要释放掉事务,如发现事
        务是只读安全的就不会存在读写依赖,所以可以释放读写冲突
        || SxactIsCommitted(conflict->sxactIn) //或者,在冲突关系链中指向本事务的
        事务已经提交、且本事务也要提交,这样就不可能构成两个连续的读写依赖
        || (conflict->sxactIn->SeqNo.lastCommitBeforeSnapshot >= PredXact-
        >LastSxactCommitSeqNo)) //或者,可能冲突的事务(conflict)的前导事务(指向本
        conflict对象的事务)的提交晚于最近提交的事务
        ReleaseRWConflict(conflict); //释放读写冲突
    conflict = nextConflict;
}

//Release all inConflicts from committed and read-only transactions. If we're
rolling back, clear them all.
conflict = (RWConflict)
    SHMQueueNext(&MySerializableXact->inConflicts, &MySerializableXact-
    >inConflicts, offsetof(RWConflictData, inLink));
while (conflict)
{
    nextConflict = (RWConflict) //inLink,表明从指向本事务的冲突关系中遍历所有指向本事
    务的冲突
    SHMQueueNext(&MySerializableXact->inConflicts, &conflict->inLink,
    offsetof(RWConflictData, inLink));

    if (!isCommit //非提交操作
        || SxactIsCommitted(conflict->sxactOut) //事务冲突链表中指向本事务的事务已经提交
        || SxactIsReadOnly(conflict->sxactOut)) //或是只读事务
        ReleaseRWConflict(conflict); //释放读写冲突

    conflict = nextConflict;
}

if (!topLevelIsDeclaredReadOnly)
{
    /*
    * Remove ourselves from the list of possible conflicts for concurrent
    * READ ONLY transactions, flagging them as unsafe if we have a
    * conflict out. If any are waiting DEFERRABLE transactions, wake them
    * up if they are known safe or known unsafe.
    */
    possibleUnsafeConflict = (RWConflict) SHMQueueNext(&MySerializableXact-
    >possibleUnsafeConflicts,
        &MySerializableXact->possibleUnsafeConflicts,
        offsetof(RWConflictData, outLink));
    while (possibleUnsafeConflict)

```





```

{
    nextConflict = (RWConflict)SHMQueueNext(&MySerializableXact->possibleUnsafeConflicts,
        &possibleUnsafeConflict->outLink, offsetof(RWConflictData, outLink));

    roXact = possibleUnsafeConflict->sxactIn; //possibleUnsafeConflict中的
    sxactIn一定是个只读事务
    Assert(MySerializableXact == possibleUnsafeConflict->sxactOut);
    Assert(SxactIsReadOnly(roXact));

    /* Mark conflicted if necessary. */
    if (isCommit
        && MyXactDidWrite
        && SxactHasConflictOut(MySerializableXact)
        //存在连续的两个相邻的读写依赖
        && (MySerializableXact->SeqNo.earliestOutConflictCommit <= roXact-
            >SeqNo.lastCommitBeforeSnapshot))
    {
        //This releases possibleUnsafeConflict (as well as all other
        possible conflicts for roXact)
        FlagSxactUnsafe(roXact);
    }
    else //指向本事务的是一个只读事务,则本事务不提交则不再会产生读写依赖,故可以释
        放;如果本事务没有发生写操作,则两个相邻的读事务也不会产生读写依赖,故可以
        释放;如果本事务和本事务指向的事务不存在读写依赖,则不会有连续的两个读写
        依赖,故可以释放;
    {
        ReleaseRWConflict(possibleUnsafeConflict);

        /* If we were the last possible conflict, flag it safe. The
        transaction can now safely release its predicate locks
        * (but that transaction's backend has to do that itself). */
        if (SHMQueueEmpty(&roXact->possibleUnsafeConflicts))
            roXact->flags |= SXACT_FLAG_RO_SAFE; //只读事务上不存在“可能的不
            安全冲突”,则只读事务必是安全的(参见9.4.2节)
    }

    //Wake up the process for a waiting DEFERRABLE transaction if we now
    know it's either safe or conflicted.
    if (SxactIsDeferrableWaiting(roXact) && (SxactIsROUnsafe(roXact) ||
        SxactIsROSafe(roXact)))
        ProcSendSignal(roXact->pid);
    possibleUnsafeConflict = nextConflict;
}

}

/*
 * Check whether it's time to clean up old transactions. This can only be

```



```

done when the last serializable transaction
* with the oldest xmin among serializable transactions completes. We then
find the "new oldest"
* xmin and purge any transactions which finished before this transaction was launched. */
needToClear = false;
if (TransactionIdEquals(MySerializableXact->xmin, PredXact->SxactGlobalXmin)
//准备清理旧的事务
{
    Assert(PredXact->SxactGlobalXminCount > 0);
    if (--(PredXact->SxactGlobalXminCount) == 0)
//清理条件：当初正在并发的事务都已经结束
    {
        SetNewSxactGlobalXmin();
        needToClear = true;
    }
}
...
/* Add this to the list of transactions to check for later cleanup. */
if (isCommit) //本事务如果提交，则把本事务加入旧事务队列，准备未来清理或汇总事务时使用
    SHMQueueInsertBefore(FinishedSerializableTransactions, &MySerializableXact->finishedLink);

if (!isCommit) //本事务回滚，则清理本事务相关内容，这时，需要释放所有本事务箭头指出导致的
                读写冲突；释放箭头指向本事务的读写冲突
    ReleaseOneSerializableXact(MySerializableXact, false, false);
//另外，把本事务占用的可用序列化事务的槽点归还给“PredXact->availableList”
...
if (needToClear) //清理旧事务的条件满足，开始清理
    ClearOldPredicateLocks();
...
/* Delete per-transaction lock table */
if (LocalPredicateLockHash != NULL)
{
    hash_destroy(LocalPredicateLockHash);
    LocalPredicateLockHash = NULL;
} //本会话的本事务结束，做清理工作，为下个事务做准备。注意下一个事务不一定是序列化隔离级别，
    所以注意清理的内容
}

```

在事务提交的时候，会对谓词锁进行释放，调用栈如下：

```

main()
SubPostmasterMain()
BackendRun()
PostgresMain()
    exec_simple_query()
    finish_xact_command()
    CommitTransactionCommand()
    CommitTransaction() //提交
    ResourceOwnerRelease()

```





```
ResourceOwnerReleaseInternal()
ReleasePredicateLocks() //释放谓词锁
```

在事务回滚的时候，会对谓词锁进行释放，调用栈如下：

```
main()
SubPostmasterMain()
BackendRun()
PostgresMain()
exec_simple_query()
finish_xact_command()
CommitTransactionCommand()
AbortTransaction() //回滚
ResourceOwnerRelease()
ResourceOwnerReleaseInternal()
ReleasePredicateLocks() //释放谓词锁
```

## 5. 谓词锁释放的时机

从前述的分析可以看出，谓词锁释放，主要的作用就是在释放读写冲突。而释放谓词锁，除了如图 9-25 所示的第②种情况是事务提交或回滚时需要释放谓词锁外，还有三种情况需要释放谓词锁：

- ❑ 第一种情况，图中标识为①的，确定事务是只读且安全的事务，则可以释放谓词锁，此种情况下不可能构成读写依赖。参见 9.4.2 节“Safe Snapshots”。
- ❑ 第二种情况，图中标识为③的，当要获取只读安全快照却不能获取、或者确定获取的是只读且安全的事务，则可以释放谓词锁。
- ❑ 第三种情况，图中标识为④的，对于 XA 两阶段提交事务，事务提交或回滚时需要释放谓词锁。
- ❑ 所以，从这些情况来看，谓词锁的释放，主要是事务处理完成，需要提交或回滚时进行的；而“Safe Snapshots”情况下释放是一个特例。

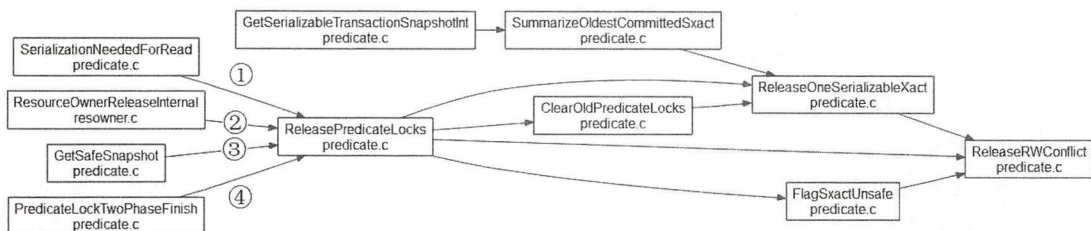


图 9-25 释放读写冲突的时机

## 6. 谓词锁的清理

因 SSI 技术的特殊性，谓词锁的清理，不是在释放谓词锁的时候进行，而是在本谓词



锁被释放的时候，去清理已经提交的、旧的谓词锁。我们对谓词锁的清理工作主要完成的任务进行分析，如下所示：

```

/*
 * Clear old predicate locks, belonging to committed transactions that are no
 * longer interesting to any in-progress transaction.
 */
static void
ClearOldPredicateLocks(void)
{
    LWLockAcquire(SerializableFinishedListLock, LW_EXCLUSIVE);
    finishedSxact = (SERIALIZABLEXACT *) //FinishedSerializableTransactions是全局变量，记录了序列化隔离级别下已经提交的事务，而这些提交的事务是按照事务提交的相对次序在此变量中有序存放的
    SHMQueueNext(FinishedSerializableTransactions, FinishedSerializableTransactions, offsetof(SERIALIZABLEXACT, finishedLink));
    LWLockAcquire(SerializableXactHashLock, LW_SHARED);
    while (finishedSxact)
    {
        SERIALIZABLEXACT *nextSxact;

        nextSxact = (SERIALIZABLEXACT *)
            SHMQueueNext(FinishedSerializableTransactions, &(finishedSxact->finishedLink), offsetof(SERIALIZABLEXACT, finishedLink));
        if (!TransactionIsValid(PredXact->SxactGlobalXmin)
            || TransactionIdPrecedesOrEquals(finishedSxact->finishedBefore,
            PredXact->SxactGlobalXmin)) //不并发则不可能产生读写依赖
        {
            LWLockRelease(SerializableXactHashLock);
            SHMQueueDelete(&(finishedSxact->finishedLink)); //老事务与当前最小的事务不并发，所以可以删除历史上的已经提交的事务
            ReleaseOneSerializableXact(finishedSxact, false, false);
            LWLockAcquire(SerializableXactHashLock, LW_SHARED);
        }
        else if (finishedSxact->commitSeqNo > PredXact->HavePartialClearedThrough
            && finishedSxact->commitSeqNo <= PredXact->CanPartialClearThrough)
        {
            /* //老事务与当前最小的事务虽然有并发的时间段，但是，在并发期间，老事务是只读数据的，没有写数据，
             * Any active transactions that took their snapshot before this
             * //所以此时也不可能产生读写依赖，所以可以清理老事务
             * transaction committed are read-only, so we can clear part of its state. */
            LWLockRelease(SerializableXactHashLock);
            if (SxactIsReadOnly(finishedSxact)) //一个只读的老事务对于活动事务的快照形成不会造成影响，可以清理
            {

```



```

        /* A read-only transaction can be removed entirely */
        SHMQueueDelete(&(finishedSxact->finishedLink));
        ReleaseOneSerializableXact(finishedSxact, false, false);
    }
    else
    {
        /*
         * A read-write transaction can only be partially cleared. We
         * need to keep the SERIALIZABLEXACT but can release the SIREAD
         locks and conflicts in.
         */
        //局部清理的一种情况
        ReleaseOneSerializableXact(finishedSxact, true, false);
    }
    ...
}
else //余下的则是不可以被清理的情况，所以不做任何工作
{
    break; /* Still interesting. */
}
finishedSxact = nextSxact;
}
LWLockRelease(SerializableXactHashLock);

//Loop through predicate locks on dummy transaction for summarized data.
LWLockAcquire(SerializablePredicateLockListLock, LW_SHARED);
predlock = (PREDICATELOCK *) //此处注意，是从OldCommittedSxact上开始遍历，即从
dummy事务开始遍历
    SHMQueueNext(&OldCommittedSxact->predicateLocks, &OldCommittedSxact-
        >predicateLocks, offsetof(PREDICATELOCK, xactLink));
while (predlock)
{
    ...
    nextpredlock = (PREDICATELOCK *)
        SHMQueueNext(&OldCommittedSxact->predicateLocks, predlock->xactLink,
            ffsetof(PREDICATELOCK, xactLink));
    ...
    canDoPartialCleanup = (predlock->commitSeqNo <= PredXact->CanPartialClearThrough);
    ...
    if (canDoPartialCleanup)
    {
        ...
        SHMQueueDelete(&(predlock->targetLink));
        SHMQueueDelete(&(predlock->xactLink));
        hash_search_with_hash_value(PredicateLockHash, &tag,
            //从PredicateLockHash中删除tag指定的对象
            PredicateLockHashCodeFromTargetHashCode(&tag, targettaghash),
            HASH_REMOVE, NULL);
        RemoveTargetIfNoLongerUsed(target, targettaghash);
    }
}

```

```

...
    }
    predlock = nextpredlock;
}
...
}

```

### 9.4.5 冲突检测

SSI 技术中除了施加谓词锁 SIRead 外，另外一个核心的功能就是冲突检测。而谓词锁的施加和释放也是为冲突检测做基础准备。

PostgreSQL 中冲突检测分为三种情况，第一种是用于检测读写依赖关系中读造成的冲突（注意做检查是在发生写操作的时刻，CheckTableForSerializableConflictIn() 函数），第二种是检测写造成的冲突（注意做检查是在发生读操作的时刻，CheckForSerializableConflictOut() 函数），第三种是在事务提交前对整个事务链进行读写冲突检测。

#### 1. 读操作冲突检测

检查读操作对于当前写操作造成的影响，是在写操作发生的时刻去做检查。当发生如图 9-26 所示的插入（索引插入、堆插入等）、删除、更新操作的时候，去检查曾经发生过的读操作对当前正在进行的写操作是否会造成读写依赖的影响。

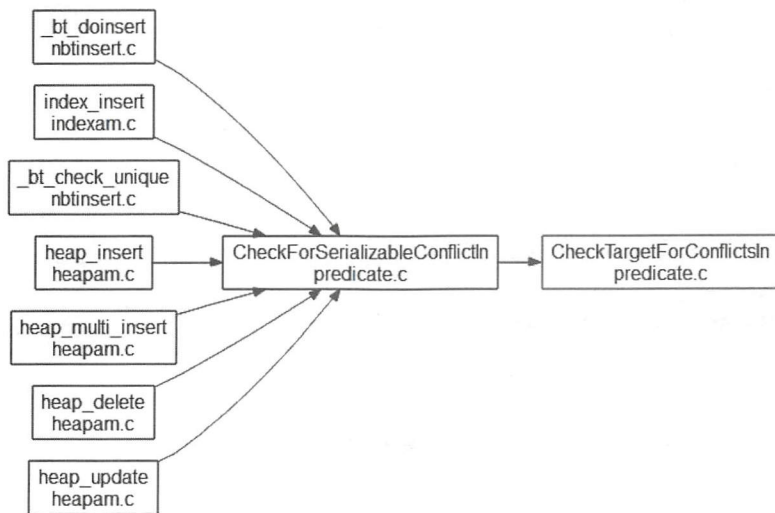


图 9-26 读操作冲突检测

以下就是写操作发生时刻，检测之前的读操作对此写操作是否可以造成读写依赖的代码。注意检测时根据入口参数的值，在三种粒度上进行。如堆插入 heap\_insert() 只在 relation 级别上进行读写依赖检查，而删除和更新操作则需要多种粒度上进行检测。



```

void
CheckForSerializableConflictIn(Relation relation, HeapTuple tuple, Buffer buffer)
{
    ...
    /* Check if someone else has already decided that we need to die */
    if (SxactIsDoomed(MySerializableXact)) //别的事务已经确定本事务需要回滚（如别的事务
    也在进行读写依赖检查，发现可以回滚其他事务以消除读写依赖，则发出让受害者回滚的指令，恰巧本事务就
    是那个受害者）
        ereport(ERROR,
                (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
                 errmsg("could not serialize access due to read/write
                 dependencies among transactions"),
                 errdetail_internal("Reason code: Canceled on identification as a
                 pivot, during conflict in checking."),
                 errhint("The transaction might succeed if retried.")));
    MyXactDidWrite = true;
    if (tuple != NULL) //如下在三种粒度上进行读写依赖检查，分别是元组、页、关系
    {
        SET_PREDICATELOCKTARGETTAG_TUPLE(targettag, relation->rd_node.dbNode,
        relation->rd_id,
        ItemPointerGetBlockNumber(&(tuple->t_self)), ItemPointerGetOffsetNum
        ber(&(tuple->t_self)));
        CheckTargetForConflictsIn(&targettag);
    }

    if (BufferIsValid(buffer))
    {
        SET_PREDICATELOCKTARGETTAG_PAGE(targettag, relation->rd_node.dbNode,
        relation->rd_id, BufferGetBlockNumber(buffer));
        CheckTargetForConflictsIn(&targettag);
    }

    SET_PREDICATELOCKTARGETTAG_RELATION(targettag, relation->rd_node.dbNode,
    relation->rd_id);
    CheckTargetForConflictsIn(&targettag);
}

```

检测之前的读操作对此写操作是否可以造成两个连续的读写依赖，真正的检测工作由 CheckTargetForConflictsIn() 函数完成，检查满足读写依赖的条件则通过 SetRWConflict() 函数标识出读写依赖。

```

static void
CheckTargetForConflictsIn(PREDICATELOCKTARGETTAG *targettag)
{
    ...
    target = (PREDICATELOCKTARGET *)
    //根据指定的对象标志targettag从PredicateLockTargetHash找出对应的谓词对象
    hash_search_with_hash_value(PredicateLockTargetHash, targettag,

```



```

targettaghash, HASH_FIND, NULL);

...
predlock = (PREDICATELOCK *) //根据指定的对象标志targettag找出该数据库对象上对应的谓词锁
    SHMQueueNext(&(target->predicateLocks), &(target->predicateLocks),
        offsetof(PREDICATELOCK, targetLink));
LWLockAcquire(SerializableXactHashLock, LW_SHARED);
while (predlock) //根据指定的对象标志targettag找出该数据库对象上对应的谓词锁，遍历每一个谓词锁。如果此循环被执行，就意味着对于对象标志targettag指定的对象上必然有过读事务，而本函数是在同一个对象上的写事务，所以必然有读写依赖存在。所以循环体内先去除读写操作是同一个事务所为的特殊情形，余者就表明存在读写依赖，所以调用FlagRWConflict设置读写依赖
{
    ...
    sxact = predlock->tag.myXact;
    if (sxact == MySerializableXact) //在同一个对象上，目前执行写操作，之前执行过读操作施加了SIREAD锁，则可以撤销SIREAD锁
    {
        if (!IsSubTransaction()
            && GET_PREDICATELOCKTARGETTAG_OFFSET(*targettag))
        {
            mypredlock = predlock; //暂存，循环结束后，释放谓词锁
            mypredlocktag = predlock->tag;
        }
    }
    else if (!SxactIsDoomed(sxact) //带有谓词锁的读事务没有被回滚
        && (!SxactIsCommitted(sxact)
            //带有谓词锁的读事务没有提交即与写事务是并发的
            || TransactionIdPrecedes(GetTransactionSnapshot()->xmin,
                sxact->finishedBefore))
        && !RWConflictExists(sxact, MySerializableXact))
        //如果已经标识过读写依赖，就不用再标识了
    {
        ...
        if (!SxactIsDoomed(sxact)
            && (!SxactIsCommitted(sxact)
                || TransactionIdPrecedes(GetTransactionSnapshot()->xmin, sxact->finishedBefore))
            && !RWConflictExists(sxact, MySerializableXact))
        {
            FlagRWConflict(sxact, MySerializableXact);
            //调用FlagRWConflict设置读写依赖
        }
    }
    ...
}
predlock = nextpredlock;
}

...
if (mypredlock != NULL) //在同一个对象上，目前执行写操作，之前执行过读操作施加了SIREAD锁，则可以撤销SIREAD锁
{
    ...

```





```

        errdetail_internal("Reason code: Canceled on identification as a
        pivot, during conflict out checking."),
        errhint("The transaction might succeed if retried."));
    }
...
htsvResult = HeapTupleSatisfiesVacuum(tuple, TransactionXmin, buffer);
switch (htsvResult) //根据所读取的元组中的xid的值, 确定“写过”此元组的事务号, 目的是想
    确定本读事务和历史上的写事务是否存在读写依赖
{
    case HEAPTUPLE_LIVE:
        if (visible)
            return;
        xid = HeapTupleHeaderGetXmin(tuple->t_data); //计算xid的情况有多种, 不再列举
        break;
    ...}
...
sxidtag.xid = xid; //xid的值赋值给sxidtag, 表明接下来的hash搜索 (hash_search()) 是
    找的历史上写元组的事务
LWLockAcquire(SerializableXactHashLock, LW_EXCLUSIVE);
//内存中的序列化事务ID不存在, 则使用SLRU到外存找
sxid = (SERIALIZABLEXID *) //用同一个对象的xidtag找出曾经写过此提交数据项的所有的读
    事务, 如果能找到, 则意味着一定存在在同一个数据对象上的后读先写这样的读写依赖关系
    hash_search(SerializableXidHash, &sxidtag, HASH_FIND, NULL);
if (!sxid) //内存中的可串行化事务ID不存在
{
    ...
    conflictCommitSeqNo = OldSerXidGetMinConflictCommitSeqNo(xid);
    if (conflictCommitSeqNo != 0) //使用SLRU在外存找到对于xid而言有读写冲突的提交号,
        即这里已经存在一个读写倚赖了, xid事务是读写倚赖中的读事务
    {
        if (conflictCommitSeqNo != InvalidSerCommitSeqNo
            && (!SxactIsReadOnly(MySerializableXact)
                //当前读操作的事务不是只读事务, 即和xid代表的事务构成另外一个读写倚赖
                || conflictCommitSeqNo <= MySerializableXact->SeqNo.
                    lastCommitBeforeSnapshot))
            ereport(ERROR, //对于当前条件表明的情形如 “MySerializableXact-->xid--
                >conflictCommitSeqNo” 的连续两个读写倚赖情况报告错误, 相当于回滚了当前的MySerializableXact事务
                (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
                 errmsg("could not serialize access due to read/write
                 dependencies among transactions"),
                 errdetail_internal("Reason code: Canceled on conflict out
                 to old pivot %u.", xid),
                 errhint("The transaction might succeed if retried."));

        if (SxactHasSummaryConflictIn(MySerializableXact)
            //有汇总事务存在读写倚赖指向MySerializableXact
            || !SHMQueueEmpty(&MySerializableXact->inConflicts))

```



```

        //有尚且活动的事务存在读写倚赖指向MySerializableXact
        ereport(ERROR, //对于当前条件表明的情形如“其他事务--
>MySerializableXact-->xid”的连续两个读写倚赖情况报告错误，相当于回滚了当前的
MySerializableXact事务

        (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
         errmsg("could not serialize access due to read/write
         dependencies among transactions"),
         errdetail_internal("Reason code: Canceled on
         identification as a pivot, with conflict out to old committed transaction %u.", xid),
         errhint("The transaction might succeed if retried."));

        MySerializableXact->flags |= SXACT_FLAG_SUMMARY_CONFLICT_OUT;
        //本事务作为读事务指向历史上已经被汇总的写事务}

...
    }

    sxact = sxid->myXact; //能执行到这里，表明已经“可能”存在一个读写依赖（可能由当前事务
    指向sxact事务），否则，之前的代码段中必然会中断本函数返回（通过报错或执行return）。找sxact就
    是一个找写事务的过程

...
    if (sxact == MySerializableXact || SxactIsDoomed(sxact)) //去除同一个事务的情况
    {... return; }

    /* We have a conflict out to a transaction which has a conflict out to a
    summarized transaction.
    * That summarized transaction must have committed first, and we can't tell
    when it committed in relation to our
    * snapshot acquisition, so something needs to be canceled. */
    if (SxactHasSummaryConflictOut(sxact)) //表明存在一个读写倚赖由sxact事务指向汇总
    事务，形如“MySerializableXact-->sxact-->汇总事务”，所以有两种选择，一是回滚sxact，二是回
    滚MySerializableXact
    {
        if (!SxactIsPrepared(sxact)) //sxact事务尚没有准备提交，则可以回滚这个事务。否
        则，只能通过报错来回滚当前事务
        {
            sxact->flags |= SXACT_FLAG_DOOMED; //事务被回滚
            LWLockRelease(SerializableXactHashLock);
            return;
        }
        else
        {
            LWLockRelease(SerializableXactHashLock);
            ereport(ERROR, //否则，只能通过报错来回滚当前事务MySerializableXact
                (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
                 errmsg("could not serialize access due to read/write
                 dependencies among transactions"),
                 errdetail_internal("Reason code: Canceled on conflict out to

```



```

        old pivot."),
        errhint("The transaction might succeed if retried.)));
    }
}

/* If this is a read-only transaction and the writing transaction has committed,
 * and it doesn't have a rw-conflict to a transaction which committed before
 * it, no conflict. */
if (SxactIsReadOnly(MySerializableXact)
    && SxactIsCommitted(sxact) //the writing transaction has committed
    && !SxactHasSummaryConflictOut(sxact)
    && (!SxactHasConflictOut(sxact)
        || MySerializableXact->SeqNo.lastCommitBeforeSnapshot < sxact->SeqNo.
            earliestOutConflictCommit))
{
    /* Read-only transaction will appear to run first. No conflict. */
    LWLockRelease(SerializableXactHashLock);
    return;
}

if (!XidIsConcurrent(xid))
{
    /* This write was already in our snapshot; no conflict. */
    LWLockRelease(SerializableXactHashLock);
    return;
}

if (RWConflictExists(MySerializableXact, sxact))
//在读和写事务之间已经有读写倚赖，就不用再次标识出这个关系了
{
    /* We don't want duplicate conflict records in the list. */
    LWLockRelease(SerializableXactHashLock);
    return;
}

//Flag the conflict. But first, if this conflict creates a dangerous structure,
//report an error.
FlagRWConflict(MySerializableXact, sxact); //之前的代码过滤了所有可以造成的两个连续的
//读写倚赖和完全不可能造成读写倚赖的情况，余下就可以把读写依赖关系标识出来了
LWLockRelease(SerializableXactHashLock);
}

```



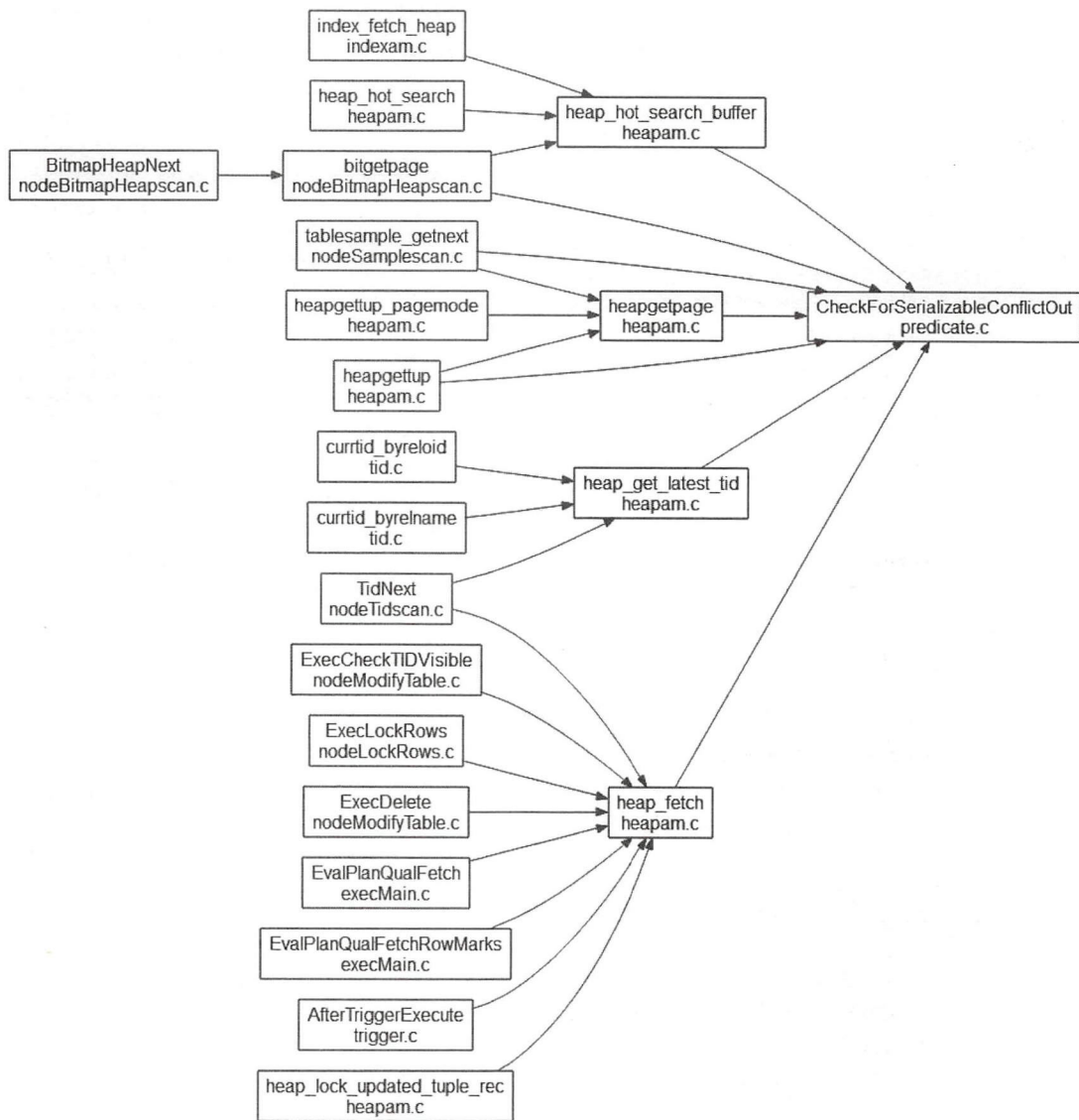


图 9-28 写操作冲突检测

### 3. 提交前的检查

对于连续两个读写依赖情况的检测，还需要在事务提交的时候进行，相关的调用上下文，参见图 9-29，相关代码如下：

```

/* If a dangerous structure is found, the pivot (the near conflict) is marked for
death, because rolling back another transaction might
* mean that we flail without ever making progress. This transaction is

```

```

committing writes, so letting it commit ensures progress.
* If we canceled the far conflict, it might immediately fail again on retry. */
void
PreCommit_CheckForSerializationFailure(void)
{...
    /* Check if someone else has already decided that we need to die */
    if (SxactIsDoomed(MySerializableXact)) //在别的事务进行读写依赖检测时，已经把当前事务当作受害者，所以本事务通过报错的方式回滚掉
    {
        LWLockRelease(SerializableXactHashLock);
        ereport(ERROR,
            (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
             errmsg("could not serialize access due to read/write
             dependencies among transactions"),
             errdetail_internal("Reason code: Canceled on identification as a
             pivot, during commit attempt."),
             errhint("The transaction might succeed if retried.")));
    }

    nearConflict = (RWConflict) //找出执向当前事务的读写依赖
        SHMQueueNext(&MySerializableXact->inConflicts, &MySerializableXact
            ->inConflicts, offsetof(RWConflictData, inLink));
    while (nearConflict) /
    {
        if (!SxactIsCommitted(nearConflict->sxactOut) //读事务没有提交
            && !SxactIsDoomed(nearConflict->sxactOut)) //读事务也没有被迫回滚
        {...
            farConflict = (RWConflict) //角色反转，读事务又作为写事务，找其对应的读写依赖
                SHMQueueNext(&nearConflict->sxactOut->inConflicts, &nearConflict-
                    >sxactOut->inConflicts,
                    offsetof(RWConflictData, inLink));
            while (farConflict)
            {
                if (farConflict->sxactOut == MySerializableXact
                    //两个事务互为指向，即两个读写依赖在两个事务之间，如图1-1的情况
                    || (!SxactIsCommitted(farConflict->sxactOut)
                        && !SxactIsReadOnly(farConflict->sxactOut)
                        && !SxactIsDoomed(farConflict->sxactOut)))
                {
                    /* Normally, we kill the pivot transaction to make sure we
                     make progress if the failing transaction is retried.
                     * However, we can't kill it if it's already prepared, so in
                     that case we commit suicide instead. */
                    if (SxactIsPrepared(nearConflict->sxactOut))
                        //形如“farConflict--> nearConflict-->MySerializableXact”的情况
                    {

```



```

        LWLockRelease (SerializableXactHashLock);
        ereport (ERROR,
            //通过抱错的方式, 回滚事务。相关技术参见7.3.4节“隐式的事务回滚”
            (errcode (ERRCODE_T_R_SERIALIZATION_FAILURE),
             errmsg ("could not serialize access due to read/
                    write dependencies among transactions"),
             errdetail_internal ("Reason code: Canceled on
                                commit attempt with conflict in from prepared pivot."),
             errhint ("The transaction might succeed if retried.")));
    }
    nearConflict->sxactOut->flags |= SXACT_FLAG_DOOMED;
    //如果不是上述情况, 则把处于pivot轴位置的事务回滚
    break;
}

farConflict = (RWConflict)
SHMQueueNext (&nearConflict->sxactOut->inConflicts, &farConflict-
               >inLink, offsetof (RWConflictData, inLink));
}

nearConflict = (RWConflict)
SHMQueueNext (&MySerializableXact->inConflicts, &nearConflict->inLink,
               offsetof (RWConflictData, inLink));
}

MySerializableXact->prepareSeqNo = ++ (PredXact->LastSxactCommitSeqNo);
MySerializableXact->flags |= SXACT_FLAG_PREPARED; //没有检测到冲突, 可以提交了, 但
未必一定能提交成功, 其他事务可能会进行检测

LWLockRelease (SerializableXactHashLock);
}

```

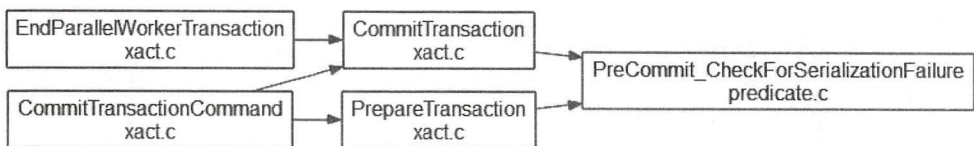


图 9-29 事务提交前的读写依赖检测

#### 4. 在两个事务间建立读写冲突的依赖

一旦检测到读写依赖, 就需要做出标识, 标识的动作通过 `FlagRWConflict()` 调用 `SetRWConflict()` 完成。其关系如图 9-30 所示, 相关上下文在前三个小节已经讲述过, 请参与, 本节将不再赘述。只是需要注意三点:

❑ `FlagRWConflict()` 调用 `SetRWConflict()` 标识一个读写依赖, 读者和写者的身份很明

确，通过入口参数指定，而且 `FlagRWConflict()` 是唯一一处表示汇总事务标志的地方。

- ❑ `SetRWConflict()` 标识一个读写依赖时，注意 `sxactIn` 和 `inConflicts` 对应的都是写者，`sxactOut` 和 `outConflicts` 对应的都是读者。
- ❑ 冲突检测函数是检测是否存在连续两个读写依赖，如果检测到，则挑选受害者回滚，这个过程基于之前标识过的读写依赖。

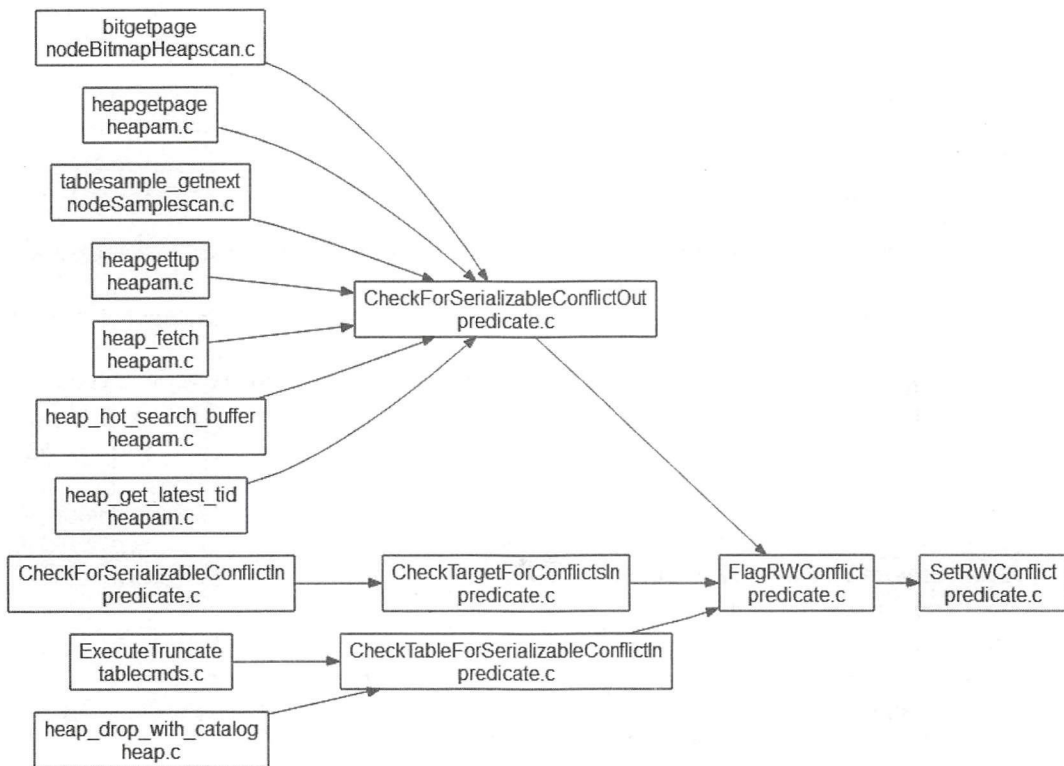


图 9-30 设置读写依赖的函数

## 5. 检查读写依赖

给定两个可串行化事务，这两个事务已经构成读写依赖，通过参数的名称指明了读者、写者的身份，判断这两个事务加入后，是否会造成两个连续的读写依赖，如果发现有这种危险结构，则回滚某个事务，这是通过如下函数完成的。该函数是检测读写依赖关系的核心函数，清晰的表明了三种情况：

- ❑ 第一种情况： $R \rightarrow W \rightarrow T2$ ， $W$  事务已经提交，且  $W$  事务指向了  $T2$  事务（ $W$  事务上有 `SXACT_FLAG_CONFLICT_OUT` 标志）即与事务  $T2$ （此时不必在意  $T2$  事务是谁，因为在释放谓词锁的时候已经为  $W$  事务打上了 `SXACT_FLAG_CONFLICT_OUT` 标志）存在读写依赖。



□ 第二种情况： $R \rightarrow W \rightarrow T_2$ ，形式同第一种，但是前提条件不同。首先， $T_2$  事务没有提交， $W$  事务先已提交。

□ 第三种情况： $T_0 \rightarrow R \rightarrow W$ ， $W$  事务先已提交。

□ 如上三种情况，如果存在连续两个读写依赖，则对应的格式为： $T_{in} \rightarrow T_{pivot} \rightarrow T_{out}$ ，其中， $T_{out}$  必定是三个事务里最先提交的（参见 9.4.2 节）。

而函数中的注释也十分地清晰易懂，结合 9.3 和 9.4.1 节中的 SSI 原理，很容易理解。

```
/*-----
 * We are about to add a RW-edge to the dependency graph - check that we don't
 * introduce a dangerous structure by doing so, and abort one of the transactions if so.
 *
 * A serialization failure can only occur if there is a dangerous structure in
 * the dependency graph:
 *
 *      Tin -----> Tpivot -----> Tout      //核心就是要查是否有两个连续的读写依赖
 *
 *          rw          rw
 *
 * Furthermore, Tout must commit first. //一个前提条件是：Tout一定是三个事务中最早提交的
 *
 * One more optimization is that if Tin is declared READ ONLY (or commits without writing) ,
 * we can only have a problem if Tout committed before Tin acquired its snapshot.
 *-----
 */
static void //注意函数的参数，进入本函数前，对于两个有待判断的事务，即已经通过reader和
            writer确定了先后次序和角色的角色
OnConflict_CheckForSerializationFailure(const SERIALIZABLEXACT *reader,
SERIALIZABLEXACT *writer)
{...
/*-----
 * Check for already-committed writer with rw-conflict out flagged
 * (conflict-flag on W means that T2 committed before W⊖):
 *
 *      R -----> W -----> T2      //第一种情况。
 *
 *          rw          rw
 *
 * That is a dangerous structure, so we must abort. (Since the writer has
 * already committed, we must be the reader)
 *-----
if (SxactIsCommitted(writer) &&
    (SxactHasConflictOut(writer) //表明在writer上已经有过读写倚赖，且writer这个事
    务当初在冲突中是读操作那方。这就表明“W --> T2”是读写倚赖
    || SxactHasSummaryConflictOut(writer))) //表明在writer上已经有过读写倚赖，且
    是writer指向汇总依赖，即前读后写。理同上。
    failure = true;

/*-----
```

⊖ The following flag actually means that the flagged transaction has a conflict out \*to a transaction which committed ahead of it\*.



```

* Check whether the writer has become a pivot with an out-conflict
* committed transaction (T2), and T2 committed first:
*
*      R -----> W -----> T2      //第二种情况。
*
*      rw          rw
*
* Because T2 must've committed first, there is no anomaly if:
//如下几种情况不会构成两个连续的读写依赖
* - the reader committed before T2      //1 读者reader先于T2提交
* - the writer committed before T2      //2 写者writer先于T2提交
* - the reader is a READ ONLY transaction and the reader was concurrent
//3 读者reader是只读事务, 且在T2提交前获取到了快照
*      with T2 (= reader acquired its snapshot before T2 committed)
//3 这意味着二者不是并发执行的
*
* We also handle the case that T2 is prepared but not yet committed here.
* In that case T2 has already checked for conflicts, so if it commits first,
* making the above conflict real, it's too late for it to abort.
*
* ----- */
if (!failure) //writer没有提交, 或者writer上没有读写依赖
{
    if (SxactHasSummaryConflictOut(writer)) //表明在writer上已经有过读写倚赖, 且是
writer指向汇总依赖, 即前读后写。
    {
        failure = true;
        conflict = NULL;
    }
    else
        conflict = (RWConflict) //遍历writer作为读操作时构成的读写依赖链表
SHMQueueNext(&writer->outConflicts, &writer->outConflicts,
offsetof(RWConflictData, outLink));
while (conflict)
{
    SERIALIZABLEXACT *t2 = conflict->sxactIn; //这是一个发生写操作的事务

    if (SxactIsPrepared(t2) //T2事务已经准备提交 (提交前已经做过PreCommit_
CheckForSerializationFailure()函数的可串行化冲突检查, 被认定是可以提交的)
&& (!SxactIsCommitted(reader) //同时, reader没有提交, 下面表明writer
没有提交, 则可能构成连续的两个读写依赖
|| t2->prepareSeqNo <= reader->commitSeqNo)
//或者, reader的提交发生在T2事务后面
&& (!SxactIsCommitted(writer) //且writer没有提交
|| t2->prepareSeqNo <= writer->commitSeqNo)
//或者, writer的提交发生在T2事务后面
&& (!SxactIsReadOnly(reader) //且reader不是只读的
|| t2->prepareSeqNo <= reader->SeqNo.
lastCommitBeforeSnapshot))
{

```



```

        failure = true;
        break;
    }
    conflict = (RWConflict)
        SHMQueueNext(&writer->outConflicts, &conflict->outLink,
            offsetof(RWConflictData, outLink));
}

/*-----
 * Check whether the reader has become a pivot with a writer that's committed
 * (or prepared):
 *
 *      T0 -----> R -----> W      //第三种情况。
 *
 *      rw          rw
 *
 * Because W must've committed first for an anomaly to occur, there is no
 * anomaly if: //不会发生异常的两种情况
 * - T0 committed before the writer
 * - T0 is READ ONLY, and overlaps the writer
 *-----*/
if (!failure && SxactIsPrepared(writer) && !SxactIsReadOnly(reader))
{
    if (SxactHasSummaryConflictIn(reader)) //T0-->Reader, 是一个读写依赖
    {
        failure = true;
        conflict = NULL;
    }
    else
    {
        conflict = (RWConflict) //遍历reader作为写操作时构成的读写依赖链表
            SHMQueueNext(&reader->inConflicts, &reader->inConflicts,
                offsetof(RWConflictData, inLink));
        while (conflict)
        {
            SERIALIZABLEXACT *t0 = conflict->sxactOut; //这是一个发生读操作的事务

            if (!SxactIsDoomed(t0) //t0事务没有被选为受害者而回滚
                && (!SxactIsCommitted(t0) //且t0事务没有提交
                    || t0->commitSeqNo >= writer->prepareSeqNo) //或者writer先提交
                && (!SxactIsReadOnly(t0) //且t0事务不是只读
                    || t0->SeqNo.lastCommitBeforeSnapshot >= writer->prepareSeqNo))
            {
                failure = true;
                break;
            }
            conflict = (RWConflict)
                SHMQueueNext(&reader->inConflicts, &conflict->inLink,
                    offsetof(RWConflictData, inLink));
        }
    }
}

```



```

    }
}

if (failure)
{
    /*
     * We have to kill a transaction to avoid a possible anomaly from occurring.
     * If the writer is us, we can just ereport() to cause a transaction abort.
     * Otherwise we flag the writer for termination, causing it to abort when
     * it tries to commit.
     * However, if the writer is a prepared transaction, already prepared,
     * we can't abort it anymore, so we have to kill the reader instead. */
    if (MySerializableXact == writer) //写事务是当前事务，则必定构成两个连续的读写依赖
    {
        ereport(ERROR,
                (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
                 errmsg("could not serialize access due to read/write
                        dependencies among transactions"),
                 errdetail_internal("Reason code: Canceled on identification
                                    as a pivot, during write."),
                 errhint("The transaction might succeed if retried.")));
    }
    else if (SxactIsPrepared(writer)) //写事务通过了读写依赖检查，因不存在读写依赖而
    可以准备提交
    {
        /* if we're not the writer, we have to be the reader */
        Assert(MySerializableXact == reader);
        ereport(ERROR,
                (errcode(ERRCODE_T_R_SERIALIZATION_FAILURE),
                 errmsg("could not serialize access due to read/write
                        dependencies among transactions"),
                 errdetail_internal("Reason code: Canceled on conflict out to
                                    pivot %u, during read.", writer->topXid),
                 errhint("The transaction might succeed if retried.")));
    }
    writer->flags |= SXACT_FLAG_DOOMED; //如果不能像前面两种情况直接通过报错（ERROR
    级别）而回滚事务，再标识这个事务需要被回滚
}
}

```

## 6. 读写冲突释放

一个读写依赖的释放很简单，代码如下，需要掌握的点是注意释放后的依赖关系即冲突（conflict）要归还会事务冲突池的可用列表，以备再次被使用。

```
static void
```



```

ReleaseRWConflict(RWConflict conflict) //释放一个读写冲突，就是把表示冲突的对象去除。此对象
逻辑上是“一边两点”
{
    SHMQueueDelete(&conflict->inLink);
    //从队列里去掉一个inLink元素，inLink表示发出读操作得事务
    SHMQueueDelete(&conflict->outLink);
    //从队列里去掉一个outLink元素，outLink表示发出写操作的事务
    SHMQueueInsertBefore(&RWConflictPool->availableList, &conflict->outLink);
    //把outLink归还
}

```

## 9.5 隔离级别

### 9.5.1 隔离级别

PostgreSQL 的文档<sup>①</sup>非常精简地讲述了四种数据异常（参见 1.1 节）：

- ❑ Dirty Read（脏读）：A transaction reads data written by a concurrent uncommitted transaction。
- ❑ Nonrepeatable Read（不可重读读）：A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read)。
- ❑ Phantom Read（幻读）：A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction。
- ❑ Serialization Anomaly（可串行化异常）：The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time。

PostgreSQL 的对异常的解决，可采用的隔离级别如表 9-5 所示。

表 9-5 PostgreSQL 支持的隔离级别与异常现象的关系

	Dirty Read 脏读	Nonrepeatable Read 不可重复读	Phantom Read 幻象	Serialization Anomaly 可串行化异常
Read uncommitted	标准允许发生但 PG 不支持脏读	允许发生	允许发生	允许发生
Read committed	不允许发生	允许发生	允许发生	允许发生
Repeatable read	不允许发生	不允许发生	标准允许发生但 PG 不支持幻象	允许发生
Serializable	不允许发生	不允许发生	不允许发生	不允许发生

① <https://www.postgresql.org/docs/9.6/static/transaction-iso.html#XACT-SERIALIZABLE>。

PostgreSQL 的代码定义了如下四种隔离级别。

```
#define XACT_READ_UNCOMMITTED 0
#define XACT_READ_COMMITTED 1
#define XACT_REPEATABLE_READ 2
#define XACT_SERIALIZABLE 3
```

一些时候，使用如下的宏帮助判断隔离级别。当“IsolationUsesXactSnapshot()”的结果值为 FALSE 时，表示隔离级别是已提交读或未提交读，但是，PostgreSQL 只实现了已提交读的语义，没有实现未提交读的语义。

```
/*
 * We implement three isolation levels internally.
 * The two stronger ones use one snapshot per database transaction;
 * the others use one snapshot per statement.
 * Serializable uses predicate locks in addition to snapshots.
 * These macros should be used to check which isolation level is selected.
 */
#define IsolationUsesXactSnapshot() (XactIsoLevel >= XACT_REPEATABLE_READ)
//可重复读与可串行化
#define IsolationIsSerializable() (XactIsoLevel == XACT_SERIALIZABLE)
//可串行化
```

## 9.5.2 各种隔离级别的实现

对于可串行化的实现，本章 9.3、9.4 节重点作了讲述，接下来分析 PostgreSQL 使用 MVCC 技术是怎么实现其他隔离级别的。

### 1. 可重复读的实现

在可重复读隔离级别下，PostgreSQL 获取快照，然后在事务内部一直使用同一个快照，这样所有的读操作都使用同一个快照，避免了这个事务中间某个时间点上，其他事务提交的数据对本事务造成影响。如下代码分析，重点在于 FirstSnapshotSet 和 FirstXactSnapshot 的获取以及使用上。

```
Snapshot
GetTransactionSnapshot(void) //根据隔离级别获取事务快照
{...
    if (HistoricSnapshotActive()) //有历史快照①，则使用历史上定制的快照，因为逻辑复制
    技术的主备机制不支持可串行化隔离级别②
```

① 历史快照，用于复制技术中解析WAL日志中与事务相关部分的内容。

② 技术上看，主备机制做读写分离时，会存在跨节点的读书据异常现象。在PostgreSQL以MVCC为主要的并发控制技术这种机制下，主备之间主写备读的使用方式同样会出现写偏序异常。解决的办法是在备机的读操作的事务信息需要告知主机，由主机统一协调（协调的方式类似SSI技术），这样才能保持真正的数据一致性。这是未来PostgreSQL需要弥补的一点。



```

{
    Assert(!FirstSnapshotSet);
    return HistoricSnapshot;
}

if (!FirstSnapshotSet) //是否使用同一个快照？如果有FirstXactSnapshot，则使用同一个不再创建，以保证整个事务块内看到的都是一致的数据
{
    ...
    //注意下面对于不同隔离级别的快照的获取，都将调用GetSnapshotData()获取快照，只是不同隔离级别需要做不同的检查，或者
    if (IsolationUsesXactSnapshot())
    //隔离级别需要大于等于可重复读“REPEATABLE READ”，即至少是可重复读或可串行化
    {
        /* First, create the snapshot in CurrentSnapshotData */
        if (IsolationIsSerializable()) //隔离级别是可串行化
            CurrentSnapshot = GetSerializableTransactionSnapshot⊖(&CurrentSnapshotData); //间接调用GetSnapshotData()获取快照
        else
            CurrentSnapshot = GetSnapshotData(&CurrentSnapshotData);
        //隔离级别是可重复读。直接调用GetSnapshotData()获取快照
        /* Make a saved copy */
        CurrentSnapshot = CopySnapshot(CurrentSnapshot);
        FirstXactSnapshot = CurrentSnapshot;
        ...
    }
    else
        CurrentSnapshot = GetSnapshotData(&CurrentSnapshotData);
        //隔离级别是已提交读“READ COMMITTED”
        /* Don't allow catalog snapshot to be older than xact snapshot. */
        CatalogSnapshotStale = true;
        FirstSnapshotSet = true; //完成一个事务内第一次获取快照的任务
        return CurrentSnapshot;
    }

    if (IsolationUsesXactSnapshot()) //第一次之后再次获取快照的可重复读和可串行化隔离级别，则不用再生成新的快照，使用老的快照即可。此处是区别于已提交读隔离级别的重要点之一，在一个事务块内的多个SQL语句是否使用“同一个”快照（FirstXactSnapshot），是可重复读（包括可串行化）和已提交读之间的最大差别
        return CurrentSnapshot;

    /* Don't allow catalog snapshot to be older than xact snapshot. */
    CatalogSnapshotStale = true;
    CurrentSnapshot = GetSnapshotData(&CurrentSnapshotData);
    //第一次之后再次获取快照的已提交读隔离级别，则生成新的快照。每次都生成新快照
    return CurrentSnapshot;
}

```

⊖ 参考8.7.2节对可串行化快照函数的分析，对比直接调用GetSnapshotData()实现可重复读隔离级别的差异，以此领悟SSI技术的本质。

在上级函数调用 `exec_simple_query()`，即同一个事务内部多次执行查询的时候，`snapshot` 给定的参数值为 `InvalidSnapshot`，所以通过调用 `GetTransactionSnapshot()` 和 `GetActiveSnapshot()` 获取的都是当前同一个快照。

```
void //exec_simple_query() 调用本函数时，传递参数 "PortalStart(portal, NULL, 0, InvalidSnapshot);",
PortalStart(Portal portal, ParamListInfo params, int eflags, Snapshot snapshot)
//即 "snapshot == InvalidSnapshot"
{...
    switch (portal->strategy)
    {
        case PORTAL_ONE_SELECT:
            /* Must set snapshot before starting executor. */
            if (snapshot) // snapshot给定的参数值为InvalidSnapshot时，不走此分支
                PushActiveSnapshot(snapshot);
            else //可重复读级别，"GetTransactionSnapshot()" 则一直返回的是
                "FirstXactSnapshot"，所以快照使用的是同一个
                PushActiveSnapshot(GetTransactionSnapshot());

            /* Create QueryDesc in portal's context; for the moment, set the
             destination to DestNone. */
            queryDesc = CreateQueryDesc((PlannedStmt *) linitial(portal-
                >stmts), portal->sourceText,

                GetActiveSnapshot(),
                InvalidSnapshot, None_Receiver, params, 0);

            ...
        }
    }
}
```

## 2. 已提交读的实现

在已提交读隔离级别下，PostgreSQL 获取快照，然后在事务内部使用不同的快照，这样所有的读操作都使用不同快照，在这个事务中间某个时间点上，其他事务提交的数据对本事务就会造成影响，因为后面获得的新快照能够获得之前已经提交的事务的数据。

同样的代码实现，位于 `GetTransactionSnapshot()` 函数中，根据隔离级别的设置，会执行到 “`GetSnapshotData(&CurrentSnapshotData)`” 之处，在与上一节同样的调用背景下，当被 `exec_simple_query()` 函数调用时，不断获取新的快照。

```
Snapshot
GetTransactionSnapshot(void) //根据隔离级别获取事务快照
{...
    if (!FirstSnapshotSet) //是否使用同一个快照？如果有FirstXactSnapshot，则使用同一个不
        再创建，以保证整个事务块内看到的都是一致的数据
    {...
        return CurrentSnapshot;
```



```

    }
    if (IsolationUsesXactSnapshot()) //此判断不满足已提交读隔离级别，所以不被执行，继续向下执行
        return CurrentSnapshot;
    ...
    CurrentSnapshot = GetSnapshotData(&CurrentSnapshotData);
    //第一次之后再次获取快照的已提交读隔离级别，则生成新的快照。每次都生成新快照
    return CurrentSnapshot;
}

```

另外，对于未提交读隔离级别，PostgreSQL 没有支持，这是因为 MVCC 技术使用快照技术实现隔离，天然屏蔽脏读异常现象。所以 PostgreSQL 在没有采用其他并发控制技术的情况下，是没有办法做到实现未提交读隔离级别的，这一点和基于封锁技术的并发控制技术、在元组上加短期锁允许脏读即实现未提交读隔离级别是不同的。

## 9.6 本章小结

本章讲述了 PostgreSQL 实现 MVCC 并发控制技术的相关内容，是第三部分的重点内容。

在本章开始部分，在 9.1 和 9.2 节分别讲述了 MVCC 技术的两大基础，一是快照技术二是元组的可见性判断。请注意，可见性判断针对的是元组级别，不是表对象等级别。对于元组的并发控制，除了快照外，还采用了记录锁，而记录锁的内容在第 8 章进行了讲述，本章不再提及，但是需要注意的是，记录锁在 PostgreSQL 的并发控制技术中所起的作用，仅仅是对元组级的并发操作进行抑制，而没有把他们汇集成一个内存锁表进行死锁检测等判断。所以我们说记录锁只是一个辅助实现手段，封锁技术尽管在 PostgreSQL 中被多种采用（再如元数据并发控制），但没有上升到占有主导作用的地位。

第 9.3 和 9.4 节，是本章的重点，也是 PostgreSQL 在并发控制技术中的精华部分，使用 SSI 技术实现了基于 MVCC 的真正的可串行化。所以我们用很大篇幅来讲述 SSI 技术在 PostgreSQL 中落地的方式和内容。这一部分，可以反复阅读，请注意，文中所提及的论文，更是精华，需要仔细研读。

因 9.3 和 9.4 节已经讲述了可串行化隔离级别，所以 9.5 节讲述了 PostgreSQL 所支持的另外两种隔离级别，这两种的实现较为简单，有了 9.1 和 9.2 节做基础，理解起来就较为容易，其中，明白他们实现的思路是关键：即快照在事务块内的获取是否每次都是新快照。

请注意，在掌握了 MVCC 和快照隔离的理论，并掌握了本章所介绍的相关内容后，并不意味着掌握了全部 MVCC 相关技术，读者还需要继续在实践中深入锻炼，自行掌握诸如 `heap_delete()`、`heap_update()` 等函数，确认其操作对元组的影响，并结合并发操作理解并发的情况下元组状态和快照之间的关系等相关的内容。

长路漫漫，唯有上下不断求索，不断继续深入，才能悟得其中真味。

## 第四篇

# InnoDB事务管理与并发控制源码分析

本篇从源码层面介绍 MySQL 数据库的 InnoDB 引擎的事务处理和并发控制的实现技术，在讲述源码实现的时候，结合第一篇的原理，讲述事务管理在工程实践中能够落地的内容。其中，第 10 章讲述 InnoDB 的事务系统的实现方式，包括事务的数据结构和事务处理的过程，以及为实现 ACID 所用到的 REDO 和 UNDO 相关的内容。第 11 章讲述 InnoDB 的并发控制系统的实现方式，主要是 InnoDB 的基于锁管理的并发控制技术的实现方式。第 12 章讲述 InnoDB 的 MVCC 并发控制技术。

建议读者在阅读理解本篇的时候，要结合第一篇的理论，对照第三篇的 PostgreSQL 的事务管理和并发控制实现技术的分析，进行对比掌握。



## 第 10 章

# InnoDB 事务系统的实现

本章主要讲述 InnoDB 的事务模型的实现技术，从代码实现的角度深入剖析事务相关的主要内容，但对于日志相关技术只做简略描述。

## 10.1 架构概述

### 10.1.1 事务和并发控制相关的文件

InnoDB 的代码位于 MySQL 源码包中 `storage\innobase` 目录下，这个目录下的子目录和文件较多，下面列出与事务管理和并发控制相关的文件及目录结构：

```
//与事务管理和并发控制相关的文件目录结构
storage\innobase
├─btr          //B+树索引结构，锁对象施加在索引项上面，而不是直接施加在记录/元组上面
├btr0btr.cc    //索引管理，索引的创建、释放等相关操作
|   btr0bulk.cc //索引上数据的批量操作
|   btr0cur.cc  //索引树上遍历对row进行修改等相关的辅助操作
|   btr0pcur.cc //持久游标，上层访问索引树的结构。SQL语句中（select、update和delete）
                对索引树的遍历条件所得到的一些固定属性从而决定索引树的遍历方法
|   btr0sea.cc  //索引树上的查找操作
├─data         //物理存储结构：字段和记录
|   data0data.cc //字段和元组级的一些操作
|   data0type.cc //数据类型的判断
...
├─handler      //继承MySQL的handler层
|   handler0alter.cc //ALTER TABLE接口
|   ha_innodb.cc //InnoDB作为一个插件对接MySQL Server的接口层，里面定义了诸如
                innobase_commit()和事务相关的接口
|   ha_innodb.h
```

```

...
|—lock      //存放InnoDB的锁操作相关文件，并发控制的核心
|   lock0iter.cc //锁的队列管理，初始化锁的队列和获得锁队列中的锁
|   lock0lock.cc //锁的管理，包括锁的授予、回收、死锁检测等操作，是锁操作相关的最主要的文件
|   lock0prdt.cc //谓词锁相关操作，用以实现空间索引的Next-Key locking算法
|   lock0wait.cc //锁等待相关的操作，包括因锁等待而挂起、释放、检查等操作
|
|—log       //日志管理
|   log0log.cc   //REDO日志，通过Mini-transaction的mtr_commit()把日志信息写到REDO
|               日志文件中
|   log0recv.cc  //恢复操作
|
...
|—mtr       //Mini-transaction（事务的重要子部分）处理（另外，在include目录里还有
|           mtr0mtr.ic这样的文件提供了与锁相关的一些基本操作）
|   mtr0log.cc   //Mini-transaction写日志部分
|   mtr0mtr.cc   //Mini-transaction日志首先要刷到日志缓冲，然后可把日志缓冲中的数据顺
|               序刷出到物理存储，节约了IO操作
|
...
|—page      //物理页面操作
|   page0cur.cc  //在物理页面上的遍历操作（插入、删除）
|   page0page.cc //索引的叶子节点相关操作。索引只是一棵树，叶子是物理数据
|   page0zip.cc  //页面压缩的相关操作
|
...
|—que       //Query graph
|   que0que.cc   //查询图（query graph）算法的实现，用以判断是否有环存在。在事务管理中
|               用以跟踪事务的执行过程
|
|—read      //Cursor read，一致性无锁读
|   read0read.cc //MVCC机制实现的主要代码，用快照隔离技术（InnoDB中称为ReadView）实
|               现一致性无锁读操作
|
...
|—sync      //同步操作相关内容
|   sync0arr.cc  //The wait array used in synchronization primitives
|   sync0debug.cc //Debug checks for latches
|   sync0rw.cc   //用于线程间同步的读写锁
|   sync0sync.cc //Mutex, the basic synchronization primitive
|—trx       //事务管理相关操作，事务管理机制的核心
|   trx0i_s.cc   //INFORMATION_SCHEMA里与事务和锁关联的系统表的相关操作：innodb_
|               trx、innodb_locks、innodb_lock_waits
|   trx0purge.cc //PURGE操作，用于清理旧版本数据
|   trx0rec.cc   //UNDO日志相关的操作，侧重于UNDO日志中使用到的page和record等相关的操作
|   trx0roll.cc  //事务回滚操作相关的内容
|   trx0rseg.cc  //回滚段的管理，包括创建、初始化、使用、释放等，rseg的意思是rollback segment
|   trx0sys.cc   //事务管理在系统层面的一些相关操作，如事务系统的启动、关闭等
|   trx0trx.cc   //事务管理相关的内容，如事务开始、事务提交、事务信息打印输出等动作
|   trx0undo.cc  //UNDO日志相关的操作，侧重于UNDO日志管理方面
|
...
|—row       //row的各种相关操作

```



```
...
|      row0log.cc      // Modification log for online index creation and online table rebuild
...
|      row0row.cc      //row相关的操作，如创建、格式化等（把不同数据类型的数据变成元组）
...
|      row0undo.cc     //INSERT/DELETE/UPDATE操作的逆向操作，即回滚时需要执行的操作
```

另外，storage\innobase\include 目录下还有很多和事务与并发控制相关的 \*.h 与 \*.ic 文件，本文不再一一列出，请自行查阅。

10.1.2 事务相关的整体架构

MySQL 的事务系统，根据 MySQL 的插件式体系结构（如图 10-1 所示）可以分为两层。

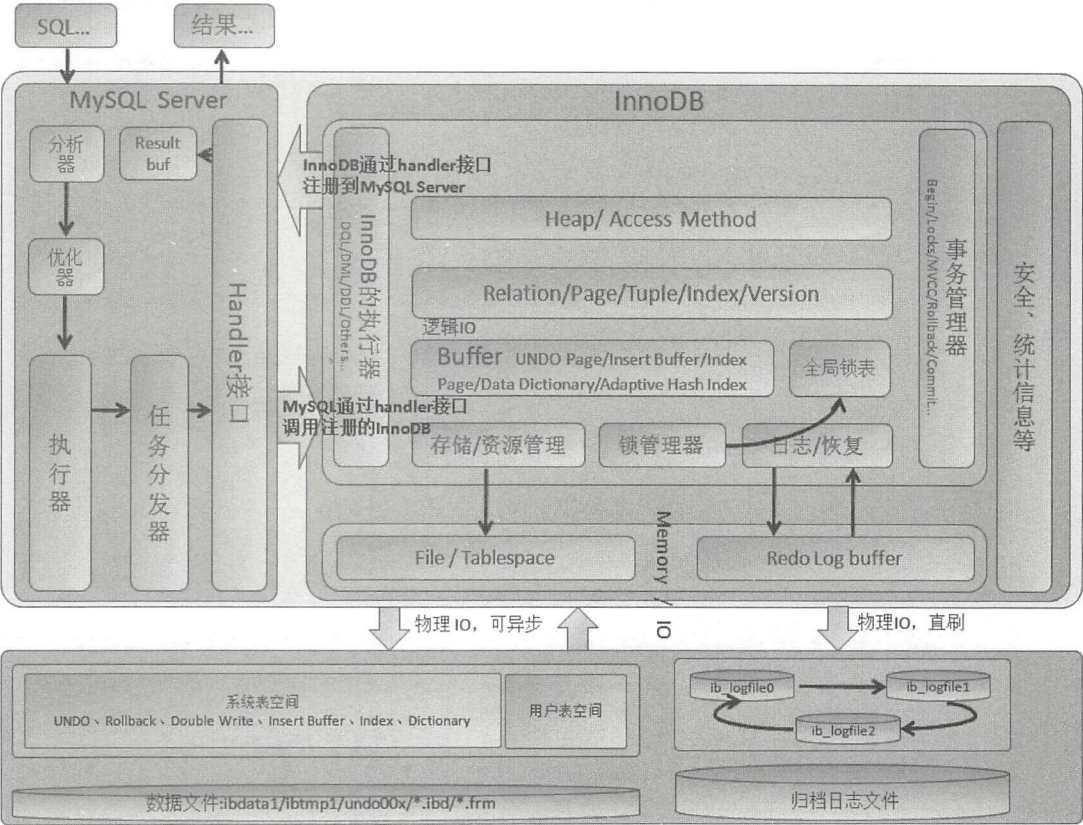


图 10-1 MySQL Server 与 InnoDB 的基本架构图

- 上层是处于运行状态的结构，下层是外存的数据存储层。
- 上部分又可以分为两部分，左面的部分是 MySQL Server 层，主要完成 SQL 的分析和优化，执行器的上层工作如连接操作在此完成；右面的部分是 InnoDB 层；MySQL Server 层和 InnoDB 层通过 Handler 接口进行交互，此接口定义了大量的各种操作，如

事务管理相关的、数据读写相关的、元信息操作如创建表等多种类型的操作。

❑ MySQL Server 层：提供了显式开启事务（`trans_begin()` 函数）、提交事务（`trans_commit()` 函数）、回滚事务（`trans_rollback()` 函数）、设置保存点（`trans_savepoint()` 函数）等操作。如图 10-1 的左上部分。

❑ InnoDB 层：通过 MySQL 的 handler 接口，调用 `innobase_init()` 注册了 InnoDB 层的事务管理相关的函数。如图 10-1 的右上部分所示。

❑ 对于 InnoDB，事务管理器和执行器是融合在一起的，共同对 SQL 语句和内部事务发挥作用。

在 InnoDB 内部，各个组件之间的关系，借用网络上的一张图（图 10-2）表述如下：

❑ 图分为两部分，左部分是内存态，右部分是外存态。

❑ 内存态主要是数据缓冲区和日志缓冲区。

❑ 外存态主要是表空间技术下管理的数据文件，另外还有元数据文件和日志文件。

❑ 各个组件之间的关系如图 10-2 所示。

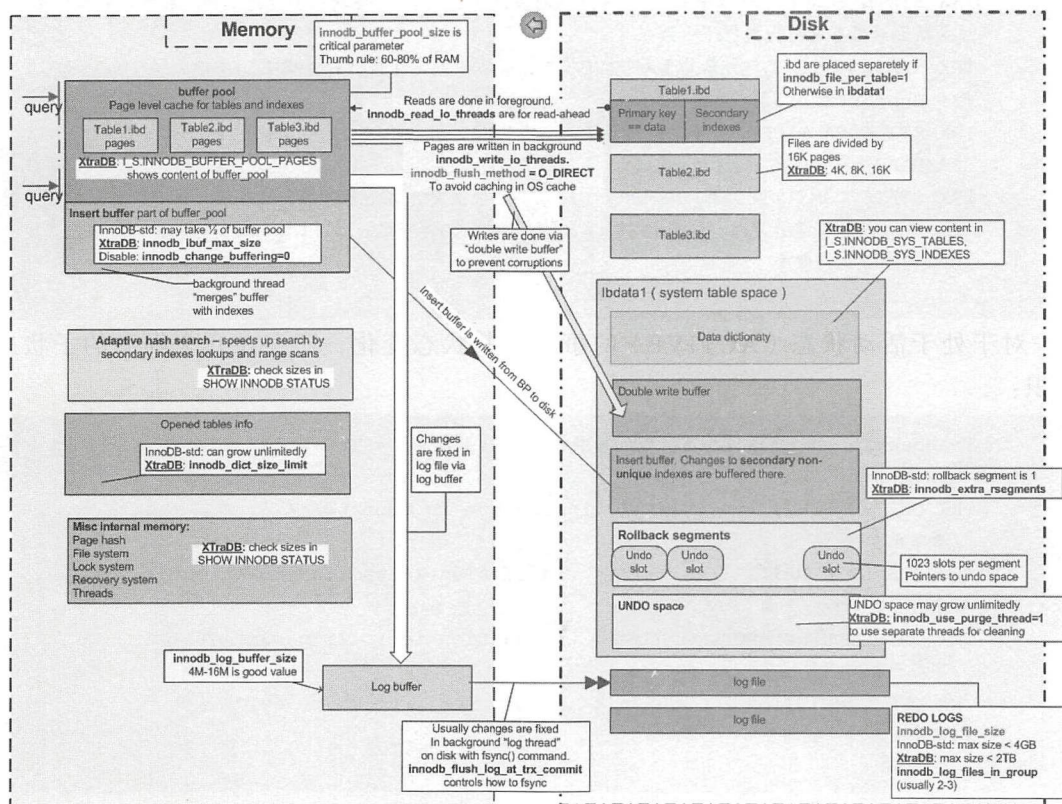


图 10-2 InnoDB 内部组件架构图<sup>⑨</sup>

⑨ 非原创，源自网络，本图权益属于原作者。



## 10.2 事务管理的基础

本节介绍 InnoDB 的事务管理的基础代码，主要涉及基本的数据结构如事务的状态和事务的数据结构。

另外，讨论了与事务紧密相关的 UNDO 日志和 REDO 日志的数据结构等。

### 10.2.1 事务状态

事务管理本质上是一个状态机，事务的生命周期里事务将处于不同的状态，可能的状态如下：

```
/** Transaction states (trx_t::state) */
enum trx_state_t {
    TRX_STATE_NOT_STARTED, // 没有事务，即事务没有开始
    /** Same as not started but with additional semantics that it was rolled back
        asynchronously the last time it was active. */
    TRX_STATE_FORCED_ROLLBACK, // 事务被回滚，这是从ACTIVE状态变迁过来的。并发机制经常会通过主动回滚事务来防止数据不一致
    TRX_STATE_ACTIVE, // 事务处于ACTIVE状态，表明事务正在执行的过程中
    /** Support for 2PC/XA */
    TRX_STATE_PREPARED, // 事务提交阶段，为支持XA，引入了2PC技术，这是2PC的第一阶段即PREPARE阶段
    TRX_STATE_COMMITTED_IN_MEMORY // 事务已经提交，这是事务的提交标识。只有事务被设置为提交标识后，才可以释放锁等资源，这是SS2PL定义的，InnoDB遵守这一点。InnoDB支持事务内存态方式提交，可以提高事务执行效率
};
```

对于处于活动状态（ACTIVE）的事务，其状态变化，还存在几个细微的子状态标识：

```
/** Transaction execution states when trx->state == TRX_STATE_ACTIVE */
enum trx_que_t {
    TRX_QUE_RUNNING, // !< transaction is running */
    // 事务正在执行
    TRX_QUE_LOCK_WAIT, // !< transaction is waiting for a lock */
    // 事务正在等待一个锁
    TRX_QUE_ROLLING_BACK, // !< transaction is rolling back */
    // 事务正在回滚过程中
    TRX_QUE_COMMITTING // !< transaction is committing */
    // 事务正在提交过程中
};
```

说明：随着事务的执行 InnoDB 事务的状态之间的转换，如图 10-3 所示，存在五种情况。

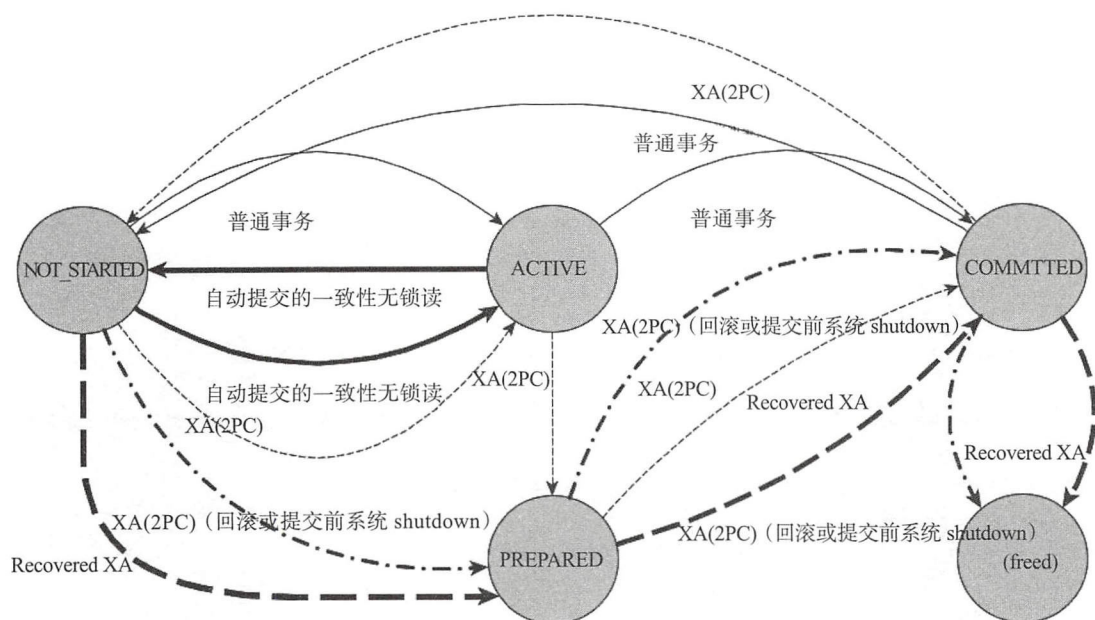


图 10-3 InnoDB 事务状态转换图

- 每种状态使用一个圆圈表示。
- 五种情况分别用不同的箭头线表示。箭头指向表示状态变迁方向。
- “(freed)”表示事务的存在是被释放不再存在，这不是一个事务状态，应对的是系统恢复的情况。
- 普通事务：TRX\_STATE\_NOT\_STARTED → ACTIVE → COMMITTED → NOT\_STARTED。
- 自动提交的一致性无锁读：TRX\_STATE\_NOT\_STARTED → ACTIVE → TRX\_STATE\_NOT\_STARTED。
- XA (2PC)：TRX\_STATE\_NOT\_STARTED → ACTIVE → PREPARED → COMMITTED → TRX\_STATE\_NOT\_STARTED。
- Recovered XA：TRX\_STATE\_NOT\_STARTED → PREPARED → COMMITTED → (freed)。
- XA (2PC)(回滚或提交前系统 shutdown)：TRX\_STATE\_NOT\_STARTED → PREPARED → (freed)。
- COMMITTED 在内存中对应的是 TRX\_STATE\_COMMITTED\_IN\_MEMORY，所以在枚举 `trx_state_t` 中不存在一个单独的表示 COMMITTED 的元素。
- 事务模型（平板事务、链式事务、嵌套事务等）的实现，严重依赖于事务的状态以及状态变迁，所以事务的状态标识是事务模型实现的一个重要元素。相关讨论，可以参考 10.3.3、10.3.4 和 10.4 节涉及事务状态的讨论。



## 10.2.2 表示事务的数据结构

事务的数据结构中，把事务的属性如隔离级别和事务的状态、与并发相关的锁和MVCC 机制的快照隔离、UNDO 日志等关联起来，主要的信息如下：

```
struct trx_t { //表示事务的数据结构
...
    bool        abort;           //事务被Abort
    trx_id_t     id;             //事务的标识，事务ID
    trx_id_t     no;             //事务的序列号
    trx_state_t  state;          //事务的状态，状态变迁参见图10-1
    ReadView*    read_view;      //活动事务的快照，与并发控制的MVCC机制关联
...
    trx_lock_t   lock;           //事务上的锁信息，与并发控制的封锁机制关联，包括了元数据锁表和记录锁表
...
    ulint        isolation_level; //事务的隔离级别
...
    bool         is_registered;   //有XA协调器注册
...
    lsn_t        commit_lsn;      //事务提交时刻的LSN，与REDO日志紧密相关，要求REDO日志
                                遵守WAL规则，事务结束前REDO日志落盘
...
    que_t*       graph;           //查询图
...
    undo_no_t    undo_no;        //下一个UNDO日志的记录号，表示事务中被修改或插入的行
...
    trx_savept_t last_sql_stat_start; //保存点，记录上一个SQL的UNDO日志的记录号
    trx_rsegs_t  rsegs;          //指向系统表空间和临时表空间，通过表空间与回滚段建立关联
...
    bool  ddl; //如果是DDL语句则内部开启一个子事务完成DDL操作。事例参见5.1.4节
    bool  internal; //是否是一个内部事务
...
    XID*   xid; // X/Open XA transaction的标识
...
};
```

从前面的数据结构分析可以看出：

- ❑ 表示事务的数据结构是 `trx_t`，在事务的结构体中，与并发控制紧密相关的是“`ReadView* read_view`”和“`trx_lock_t lock`”。
- ❑ 与事务管理相关的是“`trx_state_t state`”和“`ulint isolation_level`”等。
- ❑ 与事务管理相关的另外一个主要的结构体成员是“`trx_savept_t last_sql_stat_start`”，即事务在执行的过程中，通过保存点机制使得 SQL 语句的失败不会导致整个事务回滚，这如同使用栈（但不是栈，可以把一串 UNDO 日志视为一个逻辑上的栈）的方式实现了 5.1.2 节所举的事例功能。另外，也是“保存点 SAVEPOINT”的实现依托。

□ 与事务的回滚相关的是“`trx_rsegs_t rsegs`”→“`trx_undo_ptr_tm_redo/m_noredo`”→“`trx_rseg_t* rseg`”。而与回滚相关的主要数据结构参见 10.2.3 节。

### 10.2.3 UNDO 日志与回滚

事务通过其结构体成员 `trx_rsegs_t` 类型的 `rsegs` 与系统表空间和临时表空间等物理存储联系起来，方式如下：

```
/** Rollback segments assigned to a transaction for undo logging. */
struct trx_rsegs_t { //回滚段，每个事务都有一个回滚段
    /** undo log ptr holding reference to a rollback segment that resides in
        system/undo tablespace used for undo logging of tables that needsto be recovered on crash. */
    trx_undo_ptr_t    m_redo;    //系统的UNDO表空间

    /** undo log ptr holding reference to a rollback segment that resides in
        temp tablespace used for undo logging of tables that doesn't needto be
        recovered on crash. */
    trx_undo_ptr_t    m_noredo; //系统的临时表空间
};
```

系统表空间和临时表空间等物理存储与 UNDO 日志之间的关系如下：

```
/** Represents an instance of rollback segment along with its state variables.*/
struct trx_undo_ptr_t {
    //标识分配给事务的回滚段，这样把事务和回滚段建立起联系来。然后通过trx_rsegs_t与系统表空间
    和临时表空间等物理存储联系起来
    trx_rseg_t*    rseg;        //指向回滚段
    trx_undo_t*    insert_undo; /*!< pointer to the insert undo log, orNULL if
        no inserts performed yet */ //事务指向insert undo log
    trx_undo_t*    update_undo; /*!< pointer to the update undo log, orULL if
        no update performed yet */ //事务指向update undo log
};
```

而回滚段的信息又如下：

```
/** The rollback segment memory object */
struct trx_rseg_t {
    ulint    id;        //回滚段的标识
    ...
    ulint    space;    //回滚段的头信息在表空间中的位置，表空间标识
    ulint    page_no;  //回滚段的头信息在表空间中的位置，页号

    page_size_t    page_size; /** page size of the relevant tablespace */
    ulint            max_size; /** maximum allowed size in pages */
    ulint            curr_size; /** current size in pages */
    ...
    /** 执行UPDATE操作产生的UODO日志，包括先删除后插入的过程中产生的UODO信息，事务完成，信息
```



```

    依然被保留，用于MVCC机制下的一致性读 */
    UT_LIST_BASE_NODE_T(trx_undo_t)    update_undo_list;    /** List of update
        undo logs */
    UT_LIST_BASE_NODE_T(trx_undo_t)    update_undo_cached;/** List of update undo
        log segments cached for fast reuse */

    /* 执行INSERT操作产生的UODO日志，这些信息是临时的，事务结束后就被清理 */
    UT_LIST_BASE_NODE_T(trx_undo_t) insert_undo_list;/** List of insert undo logs */
    UT_LIST_BASE_NODE_T(trx_undo_t) insert_undo_cached;    /** List of insert
        undo log segments cached for fast reuse */

    ...
};

```

## 10.2.4 REDO 日志

REDO 日志用于记录 InnoDB 引擎的事务操作信息。这样的信息，有两个作用：

- ❑ 一是可以作为故障崩溃的时候，做系统恢复使用。以保证已经提交的事务的原子性和持久性。
- ❑ 二是可以作为流的方式实现数据同步和复制的功能（尽管 MySQL 数据同步和复制功能的主要实现方式是依据 binlog 做逻辑复制）。

REDO 日志不是本书的重点，所以本书从三个方面简要介绍如下。

### 1. 主要的数据结构

REDO 日志要使用 LSN (log sequence number) 来标识日志的顺序。

```

/* Type used for all log sequence number storage and arithmetics */
typedef uint64_t lsn_t; //一个无符号的64位整数

```

REDO 日志生成后，首先存放在一块缓存区中，称为“Redo log buffer”，然后 InnoDB 引擎以同步或异步的方式刷出 REDO 日志到外存，以实现 WAL 的功能（同步的方式才真正能实现 WAL，异步可能丢失事务造成数据不一致）。使用 REDO 日志缓存区以提高写 IO 的效率这样的实现方式，是多数数据库的选择，如 PostgreSQL 也存在类似的 REDO 日志缓存区。

```

/** Redo log buffer */
struct log_t{
    ...
    lsn_t        lsn;        /*!< log sequence number */
    ulint        buf_free;    /*在日志缓冲中，第一个空闲位置
    ...
    byte*        buf_ptr;    /* unaligned log buffer */ //REDO缓存区相关定义
    byte*        buf;        /*!< log buffer */
    ulint        buf_size;    /*!< log buffer size in bytes */
    ulint        max_buf_free; /*!< recommended maximum value of buf_free,
        after which the buffer is flushed */

```

```

...
    UT_LIST_BASE_NODE_T(log_group_t)log_groups;    //日志组

#ifdef UNIV_HOTBACKUP //定义了很多LSN即lsn_t, 以表示在缓存区中对缓存区的不同类型的操作在
缓冲区中指向的逻辑位置
...
    lsn_t      write_lsn;    /*!< last written lsn */
    lsn_t      current_flush_lsn; /*!< end lsn for the current runningwrite +
flush operation */
    lsn_t      flushed_to_disk_lsn; /*!< how far we have written the logAND
flushed to disk *///从缓存区到存储, 实现REDO的重要标识
...
    /** Fields involved in checkpoints @! */
    lsn_t      log_group_capacity; /*!< capacity of the log group; ifthe
checkpoint age exceeds this, it is
a serious error because it is possiblewe will then overwrite
log and spoilcrash recovery */
    lsn_t      max_modified_age_async;
    lsn_t      max_modified_age_sync;
    lsn_t      max_checkpoint_age_async;
    lsn_t      max_checkpoint_age;
...
    lsn_t      last_checkpoint_lsn; /*!< latest checkpoint lsn */
    lsn_t      next_checkpoint_lsn; /*!< next checkpoint lsn */
...
#endif /* !UNIV_HOTBACKUP */
...
};

```

## 2. REDO 日志文件的管理方式

REDO 日志从缓存区中被刷出后, 在外存以文件的形式存放。这样的文件, 在逻辑上归属于重做日志文件组 (group), 每个日志文件组至少有两个 REDO 日志文件, 默认的是名为 `ib_logfile0`、`ib_logfile1` 的文件。文件个数可以通过 `innodb_log_files_in_group` 参数进行调节。

日志组中每个 REDO 日志的文件大小相同, 当文件满了的时候, 会自动切换到下一个日志文件以循环使用。

在实践中, 通常为了保证安全和性能, 可为每个 REDO 日志文件设置镜像, 并分配到不同的存储介质上面。

## 3. REDO 日志与 binlog 日志的区别

我们通过表 10-1 来比较 REDO 日志与 binlog 日志的差别。



表 10-1 REDO 日志与 binlog 日志的差别表

比较项	REDO 日志	binlog 日志
记录的范围	记录事务操作过程中产生的事务日志	记录 MySQL 的所有存储引擎的日志记录 (包括 InnoDB、MyISAM 等)
记录的内容	记录的关于每个页的更改的物理情况	文件记录的格式可以为 STATEMENT 或者 ROW 或者是 MIXED
写入的时间	事务进行的过程中, 不断有 REDO 日志被写入到 REDO 日志文件中	事务提交前进行记录
记录的作用	系统故障时做恢复, 物理复制	逻辑复制

10.2.5 内部事务的处理

对于不同的操作类型, 有来自用户发出的 SQL 语句, 有来自内部的操作需要以事务的方式启动, 还有特定的情况如做系统恢复等操作, 都需要在 InnoDB 引擎内部作为事务来处理这些操作, 所以, InnoDB 区分了一些操作情况, 并提供了不同的处理方式 (如 5.1.4 节我们举例讲述过 InnoDB 在一个事务块内是怎样处理 DDL 操作的), 如表 10-2 所示。

基本上, 内部事务的处理原理和外部用户发起的事务的处理原理是一致的, 但内部发起的事务, 与 Mini Transaction 不记录日志的方式更密切, 一些相关示例可参考下一节的内容。

表 10-2 InnoDB 引擎内部对不同类型的操作处理方式表

操作 / 情况	说明	备注
DDL 操作	隐式提交之前的 SQL 操作, 然后启动一个子事务, 完成 DDL 操作后隐式提交	这两种操作, 都属于由用户发起的命令触发数据库引擎的事务过程
DML/DQL	用户的 SQL, 启动事务进行管理, 存在一个生命周期	
Purge 操作	内部特定的操作, 不占用会话进行, 是一个特殊的事务 (意味着 InnoDB 引擎内部有特殊的事务处理机制, 10.3.6 节有示例)	这两种操作, 都属于由数据库引擎发起的事务过程
系统恢复	数据库引擎完成事务相关的操作	

10.2.6 Mini-Transaction

InnoDB 的 Mini Transaction 用于实现事务的执行、REDO 日志的写入、页数据刷盘、故障恢复时页的恢复, 以实现持久性。即 REDO 日志需要基于 Mini Transaction。

InnoDB 通过 Mini Transaction 来保证并发事务的操作和数据库异常在页面级保持一致性, 即避免数据改写的过程中宕机或数据库系统宕掉后发生断页 (break page)<sup>Ⓐ</sup>问题。

Mini Transaction 的数据结构用 mtr\_t 表示, 主要标识了日志相关信息以及相关的锁信息。具体内容如下:

Ⓐ break page, 是指一个物理存储页面没有被完整的读或写, 操作到页面的一部分后, 系统宕掉后, 页面数据不一致 (要么是旧页数据, 要么是完整的新修改后的页面数据)。

```

struct mtr_t {
    /** State variables of the mtr */
    struct Impl {
        mtr_buf_t    m_memo; /** memo stack for locks etc. */
        /**管理锁信息。这是一个以栈方式管理锁的结构体
        mtr_buf_t    m_log; /** mini-transaction log */管理日志信息
        bool         m_made_dirty; /** true if mtr has made at least one buffer
                                pool page dirty */
        bool         m_inside_ibuf; /** true if inside ibuf changes */
        bool         m_modifications; /** true if the mini-transaction modified buffer pool pages */
        ib_uint32_t m_n_log_recs; /** Count of how many page initial log records
                                have been written to the mtr log */
        mtr_log_t    m_log_mode; /**Mini Transaction提供了四种类型，缺省是MTR_LOG_ALL，
        表示记录所有修改数据的操作（该不该记日志、什么要记下来对于日志技术来讲很重要，如何记日志（页面级
        还是元组级）是整体设计时就该确定的问题，影响到了后续的复制同步技术）
        ...
        fil_space_t* m_user_space; /**指向被mini-transaction修改的用户表空间
        fil_space_t* m_undo_space; /**指向被mini-transaction修改的UNDO日志表空间
        fil_space_t* m_sys_space; /**指向被mini-transaction修改的系统表空间，这三个
                                表空间是事务持久化到外存的联系纽带

        /** State of the transaction */
        mtr_state_t  m_state; /**标识Mini Transaction的状态，只有四种情况，分别是初
                                始、活动、正在提交、完成提交四种状态
        mtr_t*       m_mtr; /** Owing mini-transaction */
    };
    ...
}

```

Mini Transaction 是 InnoDB 事务处理的最核心的部分，是管理引擎内部操作（如完成系统级的加锁和释放锁、写日志等。对应有外部操作，外部操作指的是用户发起的事务相关的显示的操作）的基础。

REDO/UNDO 日志的机制需要遵循 WAL 预先日志机制，以保证日志能够在事务完成标志被设置前、日志信息能够被刷出到外存，确保持久化保存。这一点是整个日志机制的基础。尽管 Mini Transaction 涉及了 REDO 日志操作，但 Mini Transaction 只是整个事务处理过程中关键的一环，Mini Transaction 产生的日志信息，会被刷出到日志缓存区，然后在日志管理机制的统一协调下，在事务完成前，把日志缓存区中的所有日志信息刷出到外存中。更多详细讨论，请参见 10.3.3 和 10.3.4 节。

## 10.3 事务操作

MySQL Server 与 InnoDB 的事务操作，分为两个层次，内容如下：



- ❑ MySQL Server 层：提供了显式开启事务（trans\_begin() 函数）、提交事务（trans\_commit() 函数）、回滚事务（trans\_rollback() 函数）、设置保存点（trans\_savepoint() 函数）等的操作。
- ❑ InnoDB 层：通过 MySQL 的 handler 接口，调用 innobase\_init() 注册了 InnoDB 层的事务管理相关的函数，具体内容参见下一小节。

### 10.3.1 InnoDB 的初始化

InnoDB 的事务管理函数，都是在 innobase\_init() 函数中注册完成的。之后，通过函数方法完成事务的提交和回滚等操作。

```
innobase_init( //InnoDB事务相关的函数在InnoDB引擎初始化的时候，被一齐设置，之后可以被
MySQL Server通过函数指针直接调用
void      *p)    /*!< in: InnoDB handlerton */
{
...
    innobase_hton->savepoint_set = innobase_savepoint; //保存点开始
    innobase_hton->savepoint_rollback = innobase_rollback_to_savepoint; //保存点回滚
    innobase_hton->savepoint_rollback_can_release_md1 = //在回滚到保存点后是否允许安全地
    地当释放MDL锁（metadata lock）⊖
        innobase_rollback_to_savepoint_can_release_md1;
    innobase_hton->savepoint_release = innobase_release_savepoint; //释放保存点
    innobase_hton->commit = innobase_commit; //事务提交
    innobase_hton->rollback = innobase_rollback; //事务回滚
    innobase_hton->prepare = innobase_xa_prepare; //XA类型的事务的PREPARE阶段（2PC的第一阶段）
    innobase_hton->recover = innobase_xa_recover; //XA类型的事务的恢复
    innobase_hton->commit_by_xid = innobase_commit_by_xid; //XA类型的事务的第二阶段，
    执行COMMIT阶段（2PC的第二阶段）
    innobase_hton->rollback_by_xid = innobase_rollback_by_xid; //XA类型的事务的第二
    阶段，执行ROLLBACK阶段（2PC的第二阶段）
...
}
```

一旦 innobase\_hton 句柄初始化完成后，InnoDB 就可以被使用，其中事务相关的操作都会通过 innobase\_hton 中注册的函数来完成。在后面的几节中，会对 innobase\_hton 句柄中注册的事务提交、回滚等操作作详细分析。

### 10.3.2 开始事务

本节将从事务的初始化和事务启动两个方面，来探索事务模型。

#### 1. 事务初始化

事务开始，意味着首先要初始化事务，在 InnoDB 中，调用 trx\_init() 函数完成事务的

<sup>⊖</sup> 参见11.6节的内容。

初始化工作。在这个函数中，设置了事务结构体（`trx_t*trx`）的各个元素的值，如默认给定的隔离级别是“可重复读”。

图 10-4 是 `trx_init()` 函数被调用的关系图，从图中可以看出，事务被初始化，有三种途径：

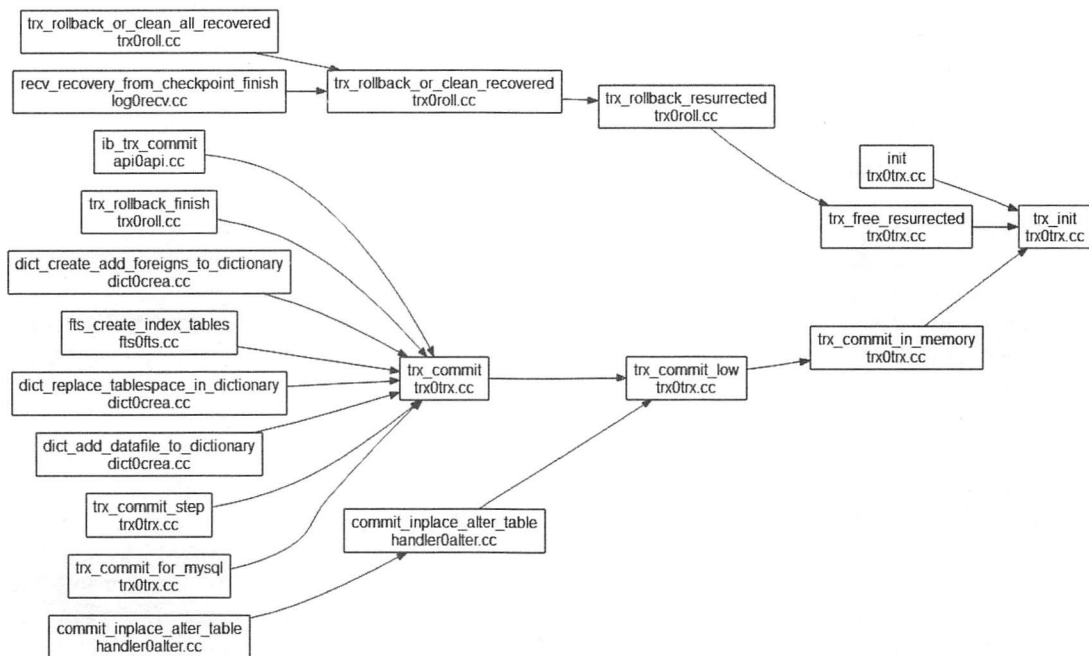


图 10-4 `trx_init()` 函数被调用的关系图

□ 一是源自于事务工厂 `TrxFactory` 调用的初始化 `init()`。

- 事务工厂就是一个事务池，预先创建、初始化好一些事务对象，避免事务对象地频繁创建和删除。
- 事务工厂在 InnoDB 被初始化的时候通过 `innobase_init()` → … → `srv_boot()` → … → `trx_pool_init()` 逐层实现多个事务的池化。
- 被池化的事务通过调用 `trx_create_low()` 函数从事务池内获取一个事务，如返回给内部的某项操作一个事务对象（是的，内部事务对象，不仅是用户 SQL 可被作为事务被数据库引擎处理，内部的一些操作也是可以当作事务来处理的。这种实现方式，不仅仅是 InnoDB 有，诸如 Informix 这样的商业数据库也存在类似的处理技术，使得一些写系统表的内部操作也事务化。这样做的好处是能够充分享受事务实现技术带来的 ACID 保障，也仅使用一套机制就满足了引擎内部和外部用户 SQL 的事务需求）。
- 事务被初始化的时候，事务的状态被设置为 `TRX_STATE_NOT_STARTED`，表示事务没有开始。
- `TrxFactory` 做初始化 `init()` 时，调用 `lock_trx_lock_list_init(&trx → lock.trx_locks)`



初始化了这个事务的已请求到的锁的 list。

○ TrxFactory 做初始化 init() 时, 还调用 lock\_trx\_alloc\_locks(trx) 为事务初始化了这个事务对应的记录锁池 (trx → lock.rec\_pool) 和表锁池 (trx → lock.table\_pool)。

□ 二是用于系统恢复时, 调用 trx\_free\_resurrected() → trx\_init() 实现事务的初始状态设置。

□ 三是事务提交或回滚之后, 在某个 session 上的事务对象被重新初始化, 以备后用 (事务实体没有被销毁而是后续备用)。如下面的 Windows 下的一段代码调用栈。

Windows 下内部事务提交或回滚之后的一段代码调用栈如下:

```

trx_init(trx_t * trx) Line 135      C++//事务的状态重新初始化, 以备后面的事务使用
trx_commit_in_memory(trx_t * trx, const mtr_t * mtr, bool serialised) Line 2117C++
trx_commit_low(trx_t * trx, mtr_t * mtr) Line 2215      C++
trx_commit(trx_t * trx) Line 2239 C++//事务提交, 提交之后要即开始一个新的事务所以事务结构
体等相关信息需要初始化
trx_commit_for_mysql(trx_t * trx) Line 2469      C++
dict_stats_fetch_from_ps(dict_table_t * table) Line 2987      C++
dict_stats_update(dict_table_t * table, dict_stats_upd_option_t stats_upd_option)
Line 3178 C++
dict_stats_init(dict_table_t * table) Line 173      C++
ha_innobase::open(const char * name, int mode, unsigned int test_if_locked) Line 5672
      C++//首次open一个表
handler::ha_open(TABLE * table_arg, const char * name, int mode, int test_if_
locked) Line 2589      C++
open_table_from_share(THD * thd, TABLE_SHARE * share, const char * alias,
unsigned int db_stat, unsigned int prgflag, unsigned int ha_open_flags, TABLE *
outparam, bool is_create_table) Line 3200      C++
open_table(THD * thd, TABLE_LIST * table_list, Open_table_context * ot_ctx) Line 3268
      C++
open_and_process_table(THD * thd, LEX * lex, TABLE_LIST * tables, unsigned int
* counter, unsigned int flags, Prelocking_strategy * prelocking_strategy, bool
has_prelocking_list, Open_table_context * ot_ctx) Line 4850      C++
open_tables(THD * thd, TABLE_LIST * * start, unsigned int * counter, unsigned int
flags, Prelocking_strategy * prelocking_strategy) Line 5469      C++
open_tables_for_query(THD * thd, TABLE_LIST * tables, unsigned int flags) Line 6206      C++
execute_sqlcom_select(THD * thd, TABLE_LIST * all_tables) Line 4870
      C++//执行SELECT命令
mysql_execute_command(THD * thd, bool first_level) Line 2564      C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305      C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command
command) Line 1251      C++
do_command(THD * thd) Line 819      C++

```

## 2. 事务启动

事务的启动函数比较简单, 根据用户的命令做简单区分, 然后设置事务的一些基本属

性信息。如对于读写事务分配回滚段，对于只读事务进行事务 ID 的分配等。事务启动的函数 `trx_start_low()` 代码分析如下：

```

trx_start_low( //开始一个事务
    trx_t*      trx,          /*!< in: transaction */
    bool        read_write)   /*!< in: true if read-write transaction */
{
    ...
    ++trx->version;
    /* Check whether it is an AUTOCOMMIT SELECT */
    trx->auto_commit = (trx->api_trx && trx->api_auto_commit) || thd_trx_is_auto_
        commit(trx->mysql_thd);

    trx->read_only = //根据环境、上下文等信息设置事务的一些属性值
        (trx->api_trx && !trx->read_write)
        || (!trx->ddl && !trx->internal && thd_trx_is_read_only(trx->mysql_thd))
        || srv_read_only_mode;
    ...
    /* The initial value for trx->no: TRX_ID_MAX is used in read_view_open_now: */
    trx->no = TRX_ID_MAX;
    ...
    /* By default all transactions are in the read-only list unless they
       are non-locking auto-commit read only transactions or background
       (internal) transactions. Note: Transactions marked explicitly as
       read only can write to temporary tables, we put those on the RO
       list too. */
    if (!trx->read_only && (trx->mysql_thd == 0 || read_write || trx->ddl))
        //非只读事务
    {
        trx->rsegs.m_redo.rseg = trx_assign_rseg_low( //对于非只读事务，分配回滚段
            srv_undo_logs, srv_undo_tablespace,
            TRX_RSEG_TYPE_REDO);

        /* Temporary rseg is assigned only if the transaction updates a temporary table */
        //对于非只读类型的事务，事务刚开始不能确定是否要用到临时表，所以临时表要使用到的临时回
        滚段暂且不分配

        trx_sys_mutex_enter();
        trx->id = trx_sys_get_new_trx_id(); //获取一个事务ID
        trx_sys->rw_trx_ids.push_back(trx->id);
        ...
        trx->state = TRX_STATE_ACTIVE; //设置事务状态为ACTIVE，由此表明在一个SESSION
            内部，事务开始了
        ...
    } else {
        trx->id = 0;
        if (!trx_is_autocommit_non_locking(trx)) { //只读事务，需要写临时表，而临时

```



```

        表需要通过事务的ID做区分
        /* If this is a read-only transaction that is writing
        to a temporary table then it needs a transaction id
        to write to the temporary table. */
        if (read_write) {
...
            trx->id = trx_sys_get_new_trx_id(); //获取一个事务ID
            trx_sys->rw_trx_ids.push_back(trx->id);
            trx_sys->rw_trx_set.insert(TrxTrack(trx->id, trx));
...
        }

        trx->state = TRX_STATE_ACTIVE; //设置事务状态为ACTIVE
    } else { //对于只读事务,没有事务ID,只标识事务的状态为开始
        ut_ad(!read_write);
        trx->state = TRX_STATE_ACTIVE; //设置事务状态为ACTIVE,每个条件判断都有同
                                        样的语句,其实可以放到条件外面只写一次
    }
}

if (trx->mysql_thd != NULL) { //设置事务的启动时间
    trx->start_time = thd_start_time_in_secs(trx->mysql_thd);
} else {
    trx->start_time = ut_time();
}
...
}

```

从图 10-5 可以看出,事务的启动函数 `trx_start_low()` 将被几种情况调用:

- ❑ 用户会话中通过执行 SQL 启动事务,调用 `trx_commit_for_mysql()` 进而调用 `trx_start_low()` 开始事务,这是主要的方式。
- ❑ 其次,数据库引擎内部,会启动一些事务操作,如:
  - 内部隐含事务操作如通过 `dict_stats_exec_sql()` 调用 `trx_commit_for_mysql()` 在数据库引擎内部执行一条 SQL 语句。
  - 内部隐含事务操作如通过 `row_create_table_for_mysql()` 调用 `trx_start_if_not_started_xa()` 在数据库引擎内部执行创建表。
  - 事务回滚阶段通过 `trx_commit_or_rollback_prepare()` 根据事务状态

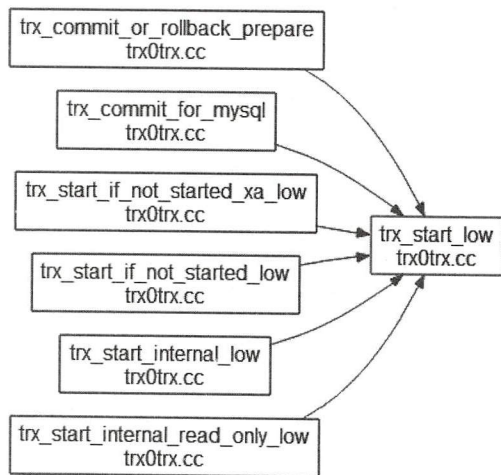


图 10-5 `trx_start_low()` 函数调用关系图

TRX\_STATE\_NOT\_STARTED 或 TRX\_STATE\_FORCED\_ROLLBACK 开启一个新事务。

- 内部隐含事务如一些 DDL 操作调用 `trx_start_for_ddl_low()`，然后再调用 `trx_start_internal_low()` 或 `trx_start_internal_read_only_low()`。`trx_start_internal_low()` 等还可以从字典表（系统表）的相关操作进行调用。

### 10.3.3 提交事务

当事务正常执行结束时，事务的提交操作，是实现事务原子性的主要操作之一，这个过程很重要。如下我们重点分析事务提交过程中一些重要环节。

#### 1. 事务提交的整体过程

InnoDB 和 MySQL Server 联合实现了事务的提交全程，相关过程如表 10-3 所示，在这个表里面，从上到下，是一个事务提交过程的栈，上面是栈顶下面是栈底，底层的函数调用了上层的函数。

表 10-3 事务提交的相关函数栈

函数名称与作用	函数过程
<code>trx_commit_in_memory</code> 在内存中标识事务完成，然后释放事务锁	1) 调用 <code>lock_trx_release_locks()</code> 释放锁 2) 断言事务的状态是在内存中已经完成（标志是上一步完成设置的） 3) 调用 <code>trx_undo_insert_cleanup()</code> 释放插入的 UNDO 日志 4) 生成新的 LSN 5) 根据 <code>innodb_flush_log_at_trx_commit</code> 参数值的情况确定是否调用 <code>trx_flush_log_if_needed()</code> 刷出日志到物理存储（即是否实现预写日志机制） 6) 如果是回滚操作，则为回滚操作设置一些值，如设置事务的状态为宏 <code>TRX_STATE_FORCED_ROLLBACK</code> 7) 重新初始化事务对象的结构体值以备再用
<code>trx_commit_low</code>	1) 释放锁 Mini-Transaction 事务提交 2) 调用 <code>trx_commit_in_memory()</code> 完成事务在内存中的提交等操作
<code>trx_commit</code>	1) 调用 <code>trx_commit_low()</code> 完成事务提交 2) 也会被 <code>trx_rollback_finish()</code> 调用，用于回滚操作
<code>trx_commit_for_mysql</code>	1) 根据事务的状态，执行不同的操作。调用 <code>trx_commit</code> 是因为事务的状态需要从 <code>TRX_STATE_ACTIVE</code> 或 <code>TRX_STATE_PREPARED</code> 转变为 <code>TRX_STATE_COMMITTED_IN_MEMORY</code>
<code>innobase_commit_low</code>	调用 <code>trx_commit_for_mysql()</code> 完成事务提交
<code>innobase_commit</code>	1) 调用 <code>innobase_commit_low()</code> 完成事务提交（设置事务提交标志并释放事务相关的锁） 2) 调用 <code>trx_commit_complete_for_mysql()</code> 刷出日志 以上是 InnoDB 层的事务提交相关代码，从上到下是从栈顶到栈底的主要函数
<code>ha_commit_low</code>	MySQL Server 层通过 <code>handle</code> 接口对底层的存储进行事务管理的操作，通过函数指针 <code>ht-&gt;commit()</code> 调用了 InnoDB 的 <code>innobase_commit()</code> 函数



(续)

函数名称与作用	函数过程
TC_LOG_DUMMY::commit	MySQL 层不提供 REDO 日志服务，但提供了一个伪接口来模拟逻辑上的日志提交操作
ha_commit_trans	MySQL 层进行的事务提交
trans_commit_stmt 或 trans_commit	MySQL 层进行的事务提交，前者是对单语句事务进行提交，后者是对多语句事务进行提交

2. 事务提交和日志刷出之间的关系

通常情况下，数据库引擎通过严格两阶段锁（SS2PL，参见第二章）来实现并发控制技术，但为了满足数据的一致性要求，事务的 ACID 特性中的一致性 C 要求事务在提交前，要先刷出日志即符合 WAL 预先日志机制。但是，InnoDB 支持事务的提交但却没有遵循 WAL 预先日志机制，这就可能带来数据不一致的问题。

如下是对 innobase\_commit() 函数的主要流程分析，从这个分析可以看出，事务先被设置提交标识，然后锁被释放。之后再执行 trx\_commit\_complete\_for\_mysql() 函数完成日志的刷出操作。所以我们说 InnoDB 不符合 WAL 预先日志机制。

```

/*****
Commits a transaction in an InnoDB database or marks an SQL statement ended.
@return 0 or deadlock error if the transaction was aborted by another higher
priority transaction. */
static
int
innobase_commit( //提交一个事务，完成事务提交的标识，然后刷出日志（如此的方式，违反了WAL预
                写日志的机制）
    handlerton*   hton,      /*!< in: InnoDB handlerton */ //InnoDB引擎的句柄
    THD*          thd,       /*!< in: MySQL thread handle of the user for whom the
transaction should be committed */ //用户会话
    bool          commit_trx) /*!< in: true - commit transaction false - the
current SQL statement ended */ //是否提交
{
    ...
    if (commit_trx //值为TRUE表示要提交事务
        || (!thd_test_options(thd, OPTION_NOT_AUTOCOMMIT | OPTION_BEGIN))) {
        ...
        innobase_commit_low(trx); //调用栈： -> trx_commit_for_mysql() -> trx_
commit(trx) -> trx_commit_low() -> trx_commit_in_memory() -> lock_trx_release_
locks(), 在lock_trx_release_locks()这个函数中执行如下重要代码：
//trx->state = TRX_STATE_COMMITTED_IN_MEMORY; 即在内存中设置事务提交已经完成的标
志，本事务的数据即刻被其他事务可见
//... 省略一些代码
//lock_release(trx); 在设置事务提交已经完成的标志后才释放锁。锁在设置提交标志后才释
放，符合SS2PL协议
    ...
    /* Now do a write + flush of logs. */
}
```

```

        if (!read_only) {
            trx_commit_complete_for_mysql(trx);
            //重要的步骤：刷出日志（刷出日志的过程可参见10.3.4节的内容）
        }
    } else { //不提交事务
        ...
    }
    ...
}

```

### 3. 为什么说 InnoDB 不符合 WAL 预写日志机制？

在第2小节中我们提出这样的调用栈：

```

innobase_commit_low(trx)
-> trx_commit_for_mysql()
   -> trx_commit(trx)
       -> trx_commit_low()
           -> trx_commit_in_memory()
               -> lock_trx_release_locks()

```

在 `lock_trx_release_locks()` 这个函数中执行如下重要代码，所幸的是，这段代码中有段很重要的注释，帮助我们回答了本标题提出的问题。

```

/*****
Releases a transaction's locks, and releases possible other transactions waiting
because of these locks. Change the state of the transaction to TRX_STATE_
COMMITTED_IN_MEMORY. *///这段注释表明了lock_trx_release_locks()函数的功能
void
lock_trx_release_locks(
/*=====*/
    trx_t*    trx)    /*!< in/out: transaction */
{...
    /* The following assignment makes the transaction committed in memory
    //这段注释很重要，需要重点理解
    and makes its changes to data visible to other transactions.
    //在内存里提交后，本事务的数据即对其他事务可见
    NOTE that there is a small discrepancy from the strict formal
    //存在的一个问题：违反了WAL预写日志的机制
    visibility rules here: a human user of the database can see
    //注释在说：即使违反了WAL预写日志的机制，InnoDB也能保证正确性
    modifications made by another transaction T even before the necessary
    log segment has been flushed to the disk. If the database happens to
    crash before the flush, the user has seen modifications from T which
    //在日志被刷出前，恰巧数据库引擎崩溃，而事务T被标识已经提交
    will never be a committed transaction. However, any transaction T2
    //即使事务T2看到了事务T崩溃前且还没有刷出的数据，事务T2要想使
    which sees the modifications of the committing transaction T, and
    //自己的修改生效，T2需要获取一个比事务T的LSN更大的一个LSN

```





```

which also itself makes modifications to the database, will get an lsn
//当系统恢复的时候, 事务T因为没有预写日志而被回滚, 而事务T2也只能回滚 (暗含之意是LSN会被用于识别并发事务的提交顺序)
larger than the committing transaction T. In the case where the log
//刷出日志时需要使用LSN判断合法性
flush fails, and T never gets committed, also T2 will never get
committed. */

/*-----*/
trx->state = TRX_STATE_COMMITTED_IN_MEMORY; //此状态一旦设置, 则本事务修改的数据则可以被其他事务所见 (此时日志还没有被刷出到外存)
...
lock_release(trx); //释放事务锁 (事务状态已经被设置, 表明提交已经完成, 本事务的数据可以被其他事务所见到, 所以可以释放锁, 这就是SS2PL中提交点应该在何时设置的技术本质)
...
}

```

这段代码的注释表明, InnoDB 知道自己事务提交的规则“可能”不符合预写日志的规则也知道这样做带来的问题 (可反复阅读上面的注释和解读), 所以也提供了相应的解决问题的方式。解决方式如下面的代码调用栈:

```

innobase_commit()
{
    innobase_commit_low(trx)
    {
        -> trx_commit_for_mysql()
        -> trx_commit(trx)
        -> trx_commit_low()
        -> trx_commit_in_memory()
        {
            -> lock_trx_release_locks()
            {
                trx->state = TRX_STATE_COMMITTED_IN_MEMORY;
                //内存中设置事务提交的标志, 本事务的数据即刻被其他事务可见
                ... //省略一些代码
                lock_release(trx); //在设置事务提交已经完成的标志后才释放锁。锁在设置提交标志后才释放, 符合SS2PL协议
            }
            ...
            lsn_t lsn = mtr->commit_lsn(); //获得最新的LSN
            if (lsn == 0) {
                /* Nothing to be done. */
            } else if (trx->flush_log_later) {
                /* Do nothing yet */
                trx->must_flush_log_later = true;
            } else if (srv_flush_log_at_trx_commit == 0)
                //innodb_flush_log_at_trx_commit参数值为0, 则不进行“预写日志”
                || thd_requested_durability(trx->mysql_thd)

```



```

== HA_IGNORE_DURABILITY) {
    /* Do nothing */
} else { //否则, innodb_flush_log_at_trx_commit参数值为
    1, 一定进行“预写日志”
    trx_flush_log_if_needed(lsn, trx);
    //第一次写日志的机会, 此时写日志, 则符合预写日志机制
}
trx->commit_lsn = lsn;
//把LSN赋值到事务结构体, 待下面执行trx_commit_complete_for_mysql(trx)时再刷出日志
}
}
...
if (!read_only) {
    trx_commit_complete_for_mysql(trx);
    //重要的步骤: 刷出日志 (刷出日志的过程可参见下一节“日志落盘”中的标题二)
    { //第二次写日志的机会, 此时写日志, 则不符合预写日志机制
        trx_flush_log_if_needed(lsn, trx) -> trx_flush_log_if_needed_low() ->
        log_write_up_to(lsn, flush);
    }
}
}
}

```

从前面的分析可以看出, InnoDB 是先设置了事务完成的状态, 然后才刷出日志 (第一次和第二次刷出日志的机会均在设置事务完成的状态之后), 所以我们说 InnoDB 不符合预写日志机制。

下面请看预写日志 (WAL) 的定义<sup>①</sup>:

In computer science, write-ahead logging (WAL) is a family of techniques for providing atomicity and durability (two of the ACID properties) in database systems.

In a system using WAL, all modifications are written to a log before they are applied. Usually both redo and undo information is stored in the log.

上面第二句话是说, 在被修改的数据被应用之前日志要刷出, 被修改的数据被应用即是数据被其他事务可见 (注意不应理解为被修改的数据被刷出到外存), 可见对应的就是事务状态被设置为已经提交。所以我们才说 InnoDB 不符合预写日志机制。

#### 4. 深入讨论: 什么是“事务提交”?

在标题三中, 我们认为 InnoDB 不符合预写日志机制, 这是基于事务提交的标志设置是在刷出日志之前完成的, 这一点, 是基于 InnoDB 的代码实现的意图来下的结论。InnoDB 认为在内存的状态设置了提交标志就算是“事务提交”完成了。

但是, 传统的数据库理论认为, “事务提交”需要符合持久性 (ACID 之 D), 也就是说, 事务的最后一个记录输出到稳定存储介质后, 才能算是“事务提交”完成了, 因为只有这样, 事务才不至于因数据库系统崩溃而发生不一致 (设置内存提交标志后、刷出日志前,

① 源自: [https://en.wikipedia.org/wiki/Write-ahead\\_logging](https://en.wikipedia.org/wiki/Write-ahead_logging).





数据库系统可能崩溃)。从这个意义上讲, InnoDB 符合预写日志机制。

### 5. 当上述问题发生时, InnoDB 怎么避免数据不一致的问题?

InnoDB 设置内存提交标志后、刷出日志前, 数据库系统可能崩溃。这种状态下:

- ❑ InnoDB 设置内存提交标志后: 事务的数据已经被其他事务可见, 存在误导其他事务发生数据不一致的可能性。
- ❑ 正常情况下, 日志被刷出: 这时, 刷出的标志是以本事务的 LSN 为刷出的位置标志, 调用 `log_write_up_to(lsn, flush)` 函数完成, 其中的参数 `lsn` 就是本事务的在日志中的最后位置, 所以一旦刷出执行成功, 则一个事务的全部信息就被完整地刷出了。此后系统崩溃, 数据库引擎做恢复时事务的完整信息存在, 则事务被恢复, 能够保证数据的一致性。这就是 ACID 中的 D 特性对 A 特性的关联保障之处。
- ❑ 刷出日志前, 数据库系统崩溃: 这时, 事务发生数据不一致的可能性就一定发生。数据库引擎做恢复时事务的完整信息不存在, 则事务被回滚, 而其他读过本事务内存态提交过的数据的事务应 LSN 值必须大于本事务的 LSN 值, 所以因事务信息不完整也会被回滚, 所以崩溃的情况下也能保证数据的一致性。

### 6. InnoDB 先提交事务后刷出日志的方式有什么好处?

- ❑ 首先, 刷出日志需要花费较多的时间, 这样会阻塞事务的提交, 导致数据库引擎整体的事务吞吐量下降。
- ❑ 其次, 事务提交提前与刷出日志正好解决了上述的问题, 能够有效提高事务吞吐量。
- ❑ PostgreSQL 数据库提供了预写日志的机制, 但 InnoDB 在使用预先日志机制的基础上, 做了改进, 区分了事务在内存态和稳定存储介质上提交的差别, 既能保证数据的最终一致性, 又有效提高了事务的并发度, 此点设计非常精妙, 值得学习和研究。

## 10.3.4 日志落盘

REDO 日志的一个重要操作, 就是确保日志能够及时写出到物理存储中 (WAL 预先日志机制), 所以需要讨论三个事情:

### 1. REDO 日志是通过什么操作写到物理存储的?

InnoDB 通过 `os_file_flush_func()` 函数实现了刷出日志到物理存储介质。注意下面代码分析中 `fsync()` 函数的使用目的。

```
os_file_flush_func(
    os_file_t    file)    /*!< in, own: handle to a file */
{
#ifdef _WIN32
    ...
    ret = FlushFileBuffers(file); //Windows系统把缓存刷出到OS
```



```

...
#else
...
    ret = os_file_fsync(file); //非Windows系统把缓存刷出到OS，中会调用系统函数fsync()⊖
    函数确保数据一定被刷出。PostgreSQL实现与此差别在于调用fsync()函数是通过参数配置确定，即为了提高性能，刷出日志到存储分为了几个粒度，fsync是最耗时但在保证REDO日志预写的最严格的保障
...
#endif /* _WIN32 */
}

```

InnoDB 调用 `os_file_flush_func()` 函数的目的有很多，如图 10-6 所示，主要分为如下几个类别：

- 日志类操作：如调用 `TruncateLogger` 类对日志文件执行 TRUNCATE 操作，调用 `log_io_complete()` 函数把 I/O 刷出到日志文件。
- 表空间类操作：如调用 `srv_undo_tablespace_create()` 函数创建一个 UNDO 表空间，调用 `fil_ibd_create()` 函数创建一个单表的表空间或常规的表空间。
- 双写缓存（doublewrite buffer）文件操作：如调用 `buf_dblwr_init_or_load_pages()` 函数写双写缓存文件。

## 2. REDO 日志刷出的调用关系

InnoDB 调用 `os_file_flush_func()` 函数实现了日志刷出的功能，其调用栈关系如下：

```

os_file_flush_func(void * file) Line 4020      C++//刷出日志到物理存储介质
fil_flush(unsigned long space_id) Line 5418C++
log_write_flush_to_disk_low() Line 1100C++    //刷出已经被写到日志文件（缓存）的日志到物理存储
log_write_up_to(unsigned __int64 lsn, bool flush_to_disk) Line 1304C++ //确保指定的
LSN的日志已经写出到日志文件（此函数还会被周期性调用刷出日志缓存区的日志信息）
trx_flush_log_if_needed_low(unsigned __int64 lsn) Line 1821C++//根据innodb_flush_
log_at_trx_commit参数确定是否刷出日志到存储
trx_flush_log_if_needed(unsigned __int64 lsn, trx_t * trx) Line 1843C++
trx_commit_complete_for_mysql(trx_t * trx) Line 2497C++
innobase_commit(handlerton * hton, THD * thd, bool commit_trx) Line 4119C++
//不是只读事务，则刷出日志到存储，属于InnoDB层的操作
ha_commit_low(THD * thd, bool all, bool run_after_commit) Line 1754      C++
TC_LOG_DUMMY::commit(THD * thd, bool all) Line 30C++
ha_commit_trans(THD * thd, bool all, bool ignore_global_read_lock) Line 1649C++
trans_commit_stmt(THD * thd) Line 460C++//属于MySQL Server层操的操作
mysql_execute_command(THD * thd, bool first_level) Line 4747C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command
command) Line 1251C++
do_command(THD * thd) Line 819C++ //栈底

```

⊖ `fsync()` 函数等待写磁盘操作结束，然后返回，这样确保数据能够持久化到存储介质而不是停留在 OS 或存储的缓冲中。这与 `sync()` 函数只把所有修改过的块缓冲区排入 OS 的写队列，然后就返回，它并不等待实际写磁盘操作结束的方式不同。所以只有前者保证日志一定能落盘，确保 WAL 的预先一定得到保证。





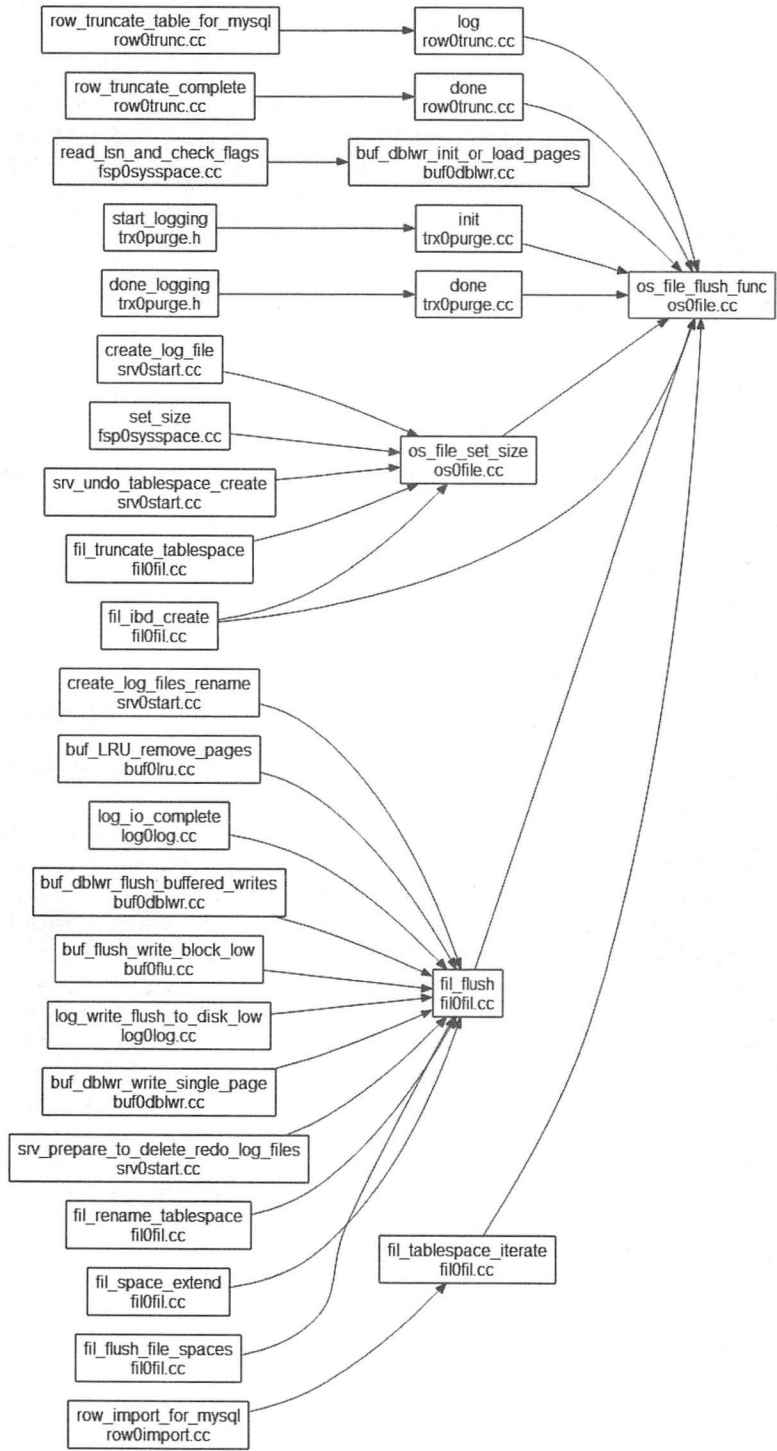


图 10-6 os\_file\_flush\_func() 函数调用关系



从这个代码调用栈和 10.3.3 节对事务提交的分析, 以及 10.3.6 节对 Mini Transaction 提交的分析中可以看出, 日志的刷出操作与 Mini Transaction 的提交不存在直接的关系, 但是与事务提交标志的设置存在重要关系 (参见 10.3.3)。

### 3. REDO 日志刷出后意味着什么?

REDO 日志刷出后, 才意味着事务真正提交成功。才能在系统崩溃的时候, 保障事务的 ACID 的 ACID 的特性 (这里和隔离性没有关系), 即:

- ❑ 确保 A 在崩溃后因能被恢复而事务完整。
- ❑ 确保 C 在崩溃后因能被恢复而数据一致。
- ❑ 确保 D 在崩溃后因日志被刷出且被恢复后的数据能持久存储。

## 10.3.5 回滚事务

当事务有异常出现时, 事务需要回滚操作, 以保证事务的原子性, 所以回滚这个过程很重要。如下我们重点分析事务回滚过程中一些重要环节, 因回滚涉及了很多 UNDO 操作和回滚段的相关内容, 非本书重点, 因此只简略涉及。

### 1. 事务回滚的整体过程

InnoDB 和 MySQL Server 联合实现了事务的回滚全程, 相关过程如表 10-4 所示, 在这个表里面, 从上到下, 是一个事务提交过程的栈, 上面是栈顶下面是栈底, 底层的函数调用了上层的函数。

表 10-4 事务回滚的相关函数栈

函数名称与作用	函数过程
trx_commit()	1) <code>trx_commit()</code> → <code>trx_commit_low()</code> → <code>trx_commit_in_memory()</code> 2) 之后的调用栈同事务提交相同, 说明回滚操作也要执行类似的过程, 回滚也可视为一种特殊的操作, 需要记录日志等信息, 所以也应该被“提交”。 3) 在 <code>trx_commit_in_memory()</code> 函数体内, 对于回滚, 特别为事务状态 <code>state</code> 设置值为 <code>TRX_STATE_FORCED_ROLLBACK</code> 。这个标志的设置意义相近于提交成功的标志设置 <code>TRX_STATE_COMMITTED_IN_MEMORY</code> , 表示本事务的数据可被其他事务所见 (注意设置这个标志是在第一次有机会进行日志刷出之后才进行的)。可参见 10.3.3 节相关讨论
trx_rollback_finish	1) 调用 <code>trx_commit()</code> 2) 调用 <code>trx → mod_tables.clear()</code> 清理被修改的表
trx_rollback_to_savepoint_low 回滚一个事务或回滚到某个保存点	1) 如果有过 <code>INSERT/UPDATE</code> 操作, 则调用 <code>que_run_threads()</code> 进行回滚操作, 后续调用关系, 参见图 10-7 2) 调用 <code>trx_rollback_finish()</code>
trx_rollback_for_mysql_low	1) 设置事务的操作信息 ( <code>trx → op_info</code> ) 2) 调用 <code>trx_rollback_to_savepoint_low()</code>
trx_rollback_low	1) 由 <code>TRX_STATE_ACTIVE</code> 状态调用 <code>trx_rollback_for_mysql_low()</code> 变迁事务状态为 <code>TRX_STATE_NOT_STARTED</code>





(续)

函数名称与作用	函数过程
trx_rollback_for_mysql	调用 trx_rollback_low() 完成事务回滚
innobase_rollback	调用 trx_rollback_for_mysql() 完成事务回滚
	以上是 InnoDB 层的事务回滚相关代码，从上到下是从栈顶到栈底的主要函数
ha_rollback_low	MySQL Server 层通过 handle 接口对底层的存储进行事务管理的操作，通过函数指针 ht → rollback() 调用了 InnoDB 的 innobase_rollback() 函数
TC_LOG_DUMMY::rollback	MySQL 层不提供 REDO 日志服务，但提供了一个伪接口来模拟逻辑上的日志回滚操作
ha_rollback_trans	MySQL 的 handle 层的事务回滚
trans_rollback	MySQL 层进行的事务回滚

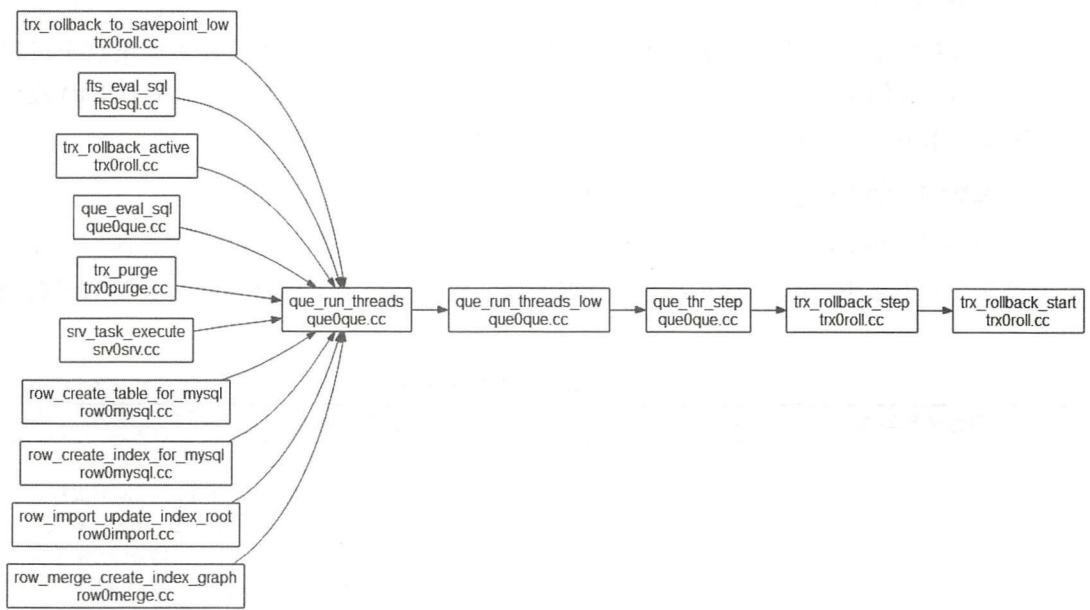


图 10-7 que\_run\_threads() 函数调用关系图

2. 事务回滚的实质过程

上一小节讲述了 InnoDB 回滚的标准过程，但回滚的全程并不仅是前面所述的内容，数据库的许多操作要被回滚，本质上是对之前所做的操作的逆向操作，如 INSERT 操作的逆向操作是删除，DELETE 操作的逆向操作是取消删除标志，UPDATE 操作的逆向操作是用旧值替换新值，这些逆向操作，被定义在了 row0undo.cc 文件中 row\_undo() 等函数中。DDL 操作不存在回滚，相应也没有对应的函数处理。

如下是一个 INSERT 操作回滚过程的一个片断。



```

mtr_t::commit() Line 443 C++ //回滚也需要Mini-Transaction的提交过程
btr_pcur_commit_specify_mtr(btr_pcur_t * pcur, mtr_t * mtr) Line 388 C++
row_undo_ins_remove_clust_rec(undo_node_t * node) Line 176 C++
row_undo_ins(undo_node_t * node, que_thr_t * thr) Line 494 C++
row_undo(undo_node_t * node, que_thr_t * thr) Line 323 C++
//调用row_undo回滚元组级别的操作 (DML造成的数据修改)
row_undo_step(que_thr_t * thr) Line 369 C++
que_thr_step(que_thr_t * thr) Line 1046 C++ //回滚框架中固定的query部分
que_run_threads_low(que_thr_t * thr) Line 1108 C++ //回滚框架中固定的query部分
que_run_threads(que_thr_t * thr) Line 1148 C++ //回滚框架中固定的query部分
trx_rollback_to_savepoint_low(trx_t * trx, trx_savept_t * savept) Line 118 C++
//回滚到底或回滚到保存点, 被MySQL Server使用; 如果是内部操作的回滚, 则被trx_rollback_to_
savepoint() 调用, 如CREATE/RENAME一个表失败, 执行过程中产生的信息需要回滚
trx_rollback_for_mysql_low(trx_t * trx) Line 180 C++ //注意函数名中带有“for_mysql”
表示函数是为MySQL Server提供服务的
trx_rollback_low(trx_t * trx) Line 213 C++
trx_rollback_for_mysql(trx_t * trx) Line 297 C++
innobase_rollback(handlerton * hton, THD * thd, bool rollback_trx) Line 4201 C++
//InnoDB内部的回滚起始入口
ha_rollback_low(THD * thd, bool all) Line 1819 C++ //MySQL Server层面的回滚
TC_LOG_DUMMY::rollback(THD * thd, bool all) Line 36 C++
ha_rollback_trans(THD * thd, bool all) Line 1889 C++
trans_rollback(THD * thd) Line 358 C++ //MySQL Server层面的回滚, 回滚起始入口
mysql_execute_command(THD * thd, bool first_level) Line 4039 C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305 C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command
command) Line 1251 C++
do_command(THD * thd) Line 819 C++

```

回滚依赖的回滚段 (struct

### 3. 事务回滚的本质

宏 TRX\_STATE\_FORCED\_ROLLBACK 描述上一个事务是被回滚掉的, 其本身的状态等同于 TRX\_STATE\_NOT\_STARTED, 表示目前没有事务处于活动状态。

在 trx\_commit\_in\_memory() 函数体内, 对于回滚操作, 事务状态 state 设置值为 TRX\_STATE\_FORCED\_ROLLBACK。这个标志的设置意义相近于提交成功的标志设置 TRX\_STATE\_COMMITTED\_IN\_MEMORY, 表示本事务的数据可被其他事务所见 (注意设置这个标志是在第一次有机会进行日志刷出之后才进行的)。如下是回滚的主要过程示意代码:

```

innobase_rollback()
{
    trx_rollback_for_mysql(trx) ->trx_rollback_low() -> trx_rollback_for_mysql_low()
    -> trx_rollback_to_savepoint_low()
}

```





```

{
    ->trx_rollback_finish()
    -> trx_commit(trx)
        -> trx_commit_low()
            ->trx_commit_in_memory() //无论是提交还是回滚，都需要调用此函数，
标志一个事务的操作正常或非正常“结束”，数据最终是一致的，本事务的数据是可以被其他事务可见的
            {
                -> lock_trx_release_locks()
                {
                    trx->state = TRX_STATE_COMMITTED_IN_MEMORY;
//内存中设置事务提交的标志，本事务的数据即刻被其他事务可见
                    ... //省略一些代码
                    lock_release(trx); //在设置事务提交已经完成的标志后才
释放锁。锁在设置提交标志后才释放，符合SS2PL协议
                }
                ...
                lsn_t lsn = mtr->commit_lsn(); //获得最新的LSN
                if (lsn == 0) {
                    /* Nothing to be done. */
                } else if (trx->flush_log_later) {
                    /* Do nothing yet */
                    trx->must_flush_log_later = true;
                } else if (srv_flush_log_at_trx_commit == 0
//innodb_flush_log_at_trx_commit参数值为0，则不进行“预写日志”
                    || thd_requested_durability(trx->mysql_thd)
                        == HA_IGNORE_DURABILITY) {
                    /* Do nothing */
                } else { //否则，innodb_flush_log_at_trx_commit参数值为
1，一定进行“预写日志”
                    trx_flush_log_if_needed(lsn, trx);
//第一次写日志的机会，此时写日志，则符合预写日志机制
                }
                trx->commit_lsn = lsn;
//把LSN赋值到事务结构体，待下面执行trx_commit_complete_for_mysql(trx)时再刷出日志
                ...
                trx_roll_savepoints_free(trx, savep);
//释放保存点（无论是提交还是回滚都需要释放保存点）
                ...
                if (trx->abort) { //如果是事务回滚
                    ...
                    trx->state = TRX_STATE_FORCED_ROLLBACK;
//如果是事务回滚，则设置事务的状态标志为回滚标志
                    ...
                } else {
                    trx->state = TRX_STATE_NOT_STARTED; //事务正常结束
                }
            }
}

```





```

...
    }
...
}
...
if (!read_only) {
    trx_commit_complete_for_mysql(trx); //重要的步骤：刷出日志（刷出日志的过程可参
    见下一节“日志落盘”中的标题二）
    { //第二次写日志的机会，此时写日志，则不符合预写日志机制（从传统数据库理论的角度看，
    不区分事务的内存态方式和外存态方式，则符合WAL）。可参见10.3.3节相关讨论
        trx_flush_log_if_needed(lsn, trx) -> trx_flush_log_if_needed_low() ->
        log_write_up_to(lsn, flush);
    }
}
}

```

### 10.3.6 Mini-Transaction 的提交

Mini-Transaction 的执行过程如下：

```

/** Commit a mini-transaction. */
void
mtr_t::commit()
{...
    m_impl.m_state = MTR_STATE_COMMITTING;
...
    if (m_impl.m_modifications
        //buffer里面的页面被修改，即有脏页存在，数据需要被刷出去
        && (m_impl.m_n_log_recs > 0 //被写到mtr日志的页面数，即存在被写到mtr的日志
            || m_impl.m_log_mode == MTR_LOG_NO_REDO)) {
        ut_ad(!srv_read_only_mode || m_impl.m_log_mode == MTR_LOG_NO_REDO);
        cmd.execute(); //执行提交，过程如下面的代码分析
    } else { //没有需要刷出的脏页，则只简单释放资源就行
        cmd.release_all(); //释放被Mini-Transaction获取到的latches和blocks
        cmd.release_resources(); //清理日志区m_log、清理memo区，标识
        Mini-Transaction完成（设置Mini-Transaction状态的值为MTR_STATE_COMMITTED）
    }
}
}

```

其中，Mini-Transaction 的执行过程如下：

```

mtr_t::Command::execute() //写REDO日志并释放资源
{...
    if (const ulint len = prepare_write()) { //准备写
        finish_write(len); //写到日志缓存中，注意REDO日志数据还没有落盘
    }
...
}

```





```

release_blocks();           //释放Mini-Transaction中的blocks (blocks是一个动态的、内
                             存缓存区, 这个区域包括多个单独的block)

...

release_latches();          //释放latch
release_resources();         //清理日志区m_log、清理memo区, 标识Mini-Transaction完成 (设置
                             Mini-Transaction状态的值为MTR_STATE_COMMITTED)
}

```

Mini-Transaction 的开始和提交的使用方式, 有多种, 但都是用于数据库引擎内部事务的相关操作, 非用户显式事务的提交或回滚。第一种是一种精简方式, 这种方式的特点在于依赖 Mini-Transaction 的相关操作, 但不生成 REDO 日志。例如, 在对 UNDO 表空间执行 TRUNCATE 操作 (PURGE 操作的一部分), 依赖 Mini-Transaction 的简单提交。

```

bool
trx_undo_truncate_tablespace(      //Truncate UNDO tablespace, reinitialize header and rseg.
    undo::Truncate*    undo_trunc) //@param[in]undo_trunc, UNDO tablespace handler
{...
    /* Step-1: Truncate tablespace. */
    success = fil_truncate_tablespace(space_id, SRV_UNDO_TABLESPACE_SIZE_IN_PAGES);
    ...

    /* Step-2: Re-initialize tablespace header.
        Avoid REDO logging as we don't want to apply the action if server
        crashes. For fix-up we have UNDO-truncate-ddl-log. */
    mtr_t      mtr;
    mtr_start(&mtr); //开始Mini-Transaction
    mtr_set_log_mode(&mtr, MTR_LOG_NO_REDO); //不记REDO日志
    fsp_header_init(space_id, SRV_UNDO_TABLESPACE_SIZE_IN_PAGES, &mtr);
    mtr_commit(&mtr); //提交Mini-Transaction。这一段, 是Mini-Transaction的一个典型用法, 进行资源管理 (释放、清理) 但不记录日志

    /* Step-3: Re-initialize rollback segment header that resides in truncated tablespaced. */
    mtr_start(&mtr); //开始Mini-Transaction
    mtr_set_log_mode(&mtr, MTR_LOG_NO_REDO); //不记REDO日志
    mtr_x_lock(fil_space_get_latch(space_id, NULL), &mtr);
    ...

    mtr_commit(&mtr); //提交Mini-Transaction。这一段, 还是Mini-Transaction的一个典型用法, 进行资源管理 (释放、清理) 但不记录日志
    ...
}

```

再举一例, InnoDB 系统启动阶段, 进行 Mini-Transaction 的开始和提交。

```

/*****
Creates and initializes the transaction system at the database creation. */
void
trx_sys_create_sys_pages(void)
{...

```



```

mtr_start(&mtr);           //即调用mtr_t::start()
trx_sysf_create(&mtr);    //在系统表空间中创建事务系统的文件页，此操作需要放到Mini-Transaction中
mtr_commit(&mtr);
}

```

### 10.3.7 Mini-Transaction 的回滚

一般而言，事务的操作，正常执行则可以提交，有异常则需要回滚，这是保证事务要么成功、要么失败的机制。所以提交操作存在对应的回滚操作。

上一节我们讨论了 Mini-Transaction 的提交操作，那么 Mini-Transaction 的回滚操作应该是什么样的呢？

翻遍 InnoDB 的代码，其实也找不到类似 “#define mtr\_commit(m) (m)->commit()” 这样的执行 Mini-Transaction 提交操作的类似的宏存在，这是在说对于 Mini-Transaction 存在提交操作但不存在回滚操作，如前所述，既然号称“事务”，就应该有提交和回滚对应，为什么 Mini-Transaction 不需要回滚操作呢？

现在正是为 Mini-Transaction 正名的时候了。

- ❑ 首先，Mini-Transaction 不是常规所说的“事务”概念，因为事务是要符合 ACID 特性的，而 Mini-Transaction 不存在“Mini 的符合 ACID”特性的特性。
- ❑ 其次，Mini-Transaction 只是用于日志和系统锁部分的管理的主要模块，属于为实现事务概念这个整体中的局部一份子。
- ❑ 最后，Mini-Transaction 的提交操作只是表明事务中的一小段工作告一段落，写日志是把这段工作记录下来不是把整个事务记录下来，是多次 Mini-Transaction 记录的日志累积构成了一个事务需要的所有的日志信息。举个例子说，如上一节对于 PURGE 过程中对 UNDO 表空间执行 TRUNCATE 操作时在 `trx_undo_truncate_tablespace()` 函数中，就存在多段 Mini-Transaction 开始和提交的操作。所以 Mini-Transaction 只是事务的一个片段。

### 10.3.8 SAVEPOINT

InnoDB 支持 SAVEPOINT，语法符合 ANSI SQL 规范，其实现方式很简单，一个 SAVEPOINT 就是一个回滚段的编号，只不过增加了一个“SAVEPOINT Name”。

#### 1. SAVEPOINT 的本质

InnoDB 使用如下两个函数，支持创建 SAVEPOINT，`trx_savepoint_for_mysql()` 函数做一些事务检查以及上下文环境处理相关的工作，然后调用 `trx_savept_take(trx)` 完成事务的创建工作。

```

/*****
Creates a named savepoint. If the transaction is not yet started, starts it.

```





```

If there is already a savepoint of the same name, this call erases that old
savepoint and replaces it with a new. Savepoints are deleted in a transaction
commit or rollback.
@return always DB_SUCCESS */
dberr_t
trx_savepoint_for_mysql(
    trx_t*      trx,          /*!< in: transaction handle */
    const char*savepoint_name, /*!< in: savepoint name */
    int64_t      binlog_cache_pos) /*!< in: MySQL binlog cache position
    corresponding to thisconnection at the time of thesavepoint */
{
    ...
    trx_start_if_not_started_xa(trx, false); //如果没有开始一个事务, 则开始一个新事务
    savep = trx_savepoint_find(trx, savepoint_name); //查找指定名称的savepoint
    if (savep) { //存在指定名称的savepoint则先释放
        /* There is a savepoint with the same name: free that */
        UT_LIST_REMOVE(trx->trx_savepoints, savep);
        ut_free(savep->name);
        ut_free(savep);
    }

    /* Create a new savepoint and add it as the last in the list */
    savep = static_cast<trx_named_savept_t*>(ut_malloc_nokey(sizeof(*savep)));
    savep->name = mem_strdup(savepoint_name);
    savep->savept = trx_savept_take(trx); //创建一个trx_savept_t对象
    savep->mysql_binlog_cache_pos = binlog_cache_pos;
    UT_LIST_ADD_LAST(trx->trx_savepoints, savep);
    return(DB_SUCCESS);
}

```

函数 `trx_savept_take(trx)` 用以完成事务的创建工作, 返回 `trx_savept_t` 定义的值, 此值包括一个字符串类型的名字和数值型的回滚段编号。

```

/*****
Returns a transaction savepoint taken at this point in time.
@return savepoint */
trx_savept_t
trx_savept_take(
    trx_t*      trx) /*!< in: transaction */{
    trx_savept_t  savept;
    savept.least_undo_no = trx->undo_no; //如此简单的一个赋值, 即做一个位置标识, 且是回
    滚段的标识, 可以看出, InnoDB的Savepoint依赖回归段实现, 十分简单简洁
    return(savept);
}

```

而 `least_undo_no` 和 `undo_no` 的定义如下, 即一个数值型的编号表示。

```

/** Undo number */
typedef ib_id_tundo_no_t; //一个数值型的编号
/** Transaction savepoint */

```



```
struct trx_savept_t{
    undo_no_t least_undo_no;    /*!< least undo number to undo */
};
```

这表明, SAVEPOINT 的本质, 在 InnoDB 中就是事务过程中不断产生的多个回滚段的命名手段, 使得回滚段可被用户操作 (命名、回滚到某个有名字的回滚段上)。所以, 理解 InnoDB 的 SAVEPOINT, 需要先理解 InnoDB 的 UODO 日志相关内容。

## 2. SAVEPOINT 的相关操作

InnoDB 支持的 SAVEPOINT 相关的操作如表 10-5 所示, 这些函数非常简单, 不再详述。

表 10-5 SAVEPOINT 的相关函数表

函数名称	函数作用
trx_savepoint_for_mysql	创建 SAVEPOINT
trx_rollback_to_savepoint	回滚到一个 SAVEPOINT
trx_savepoint_find	查找 SAVEPOINT
trx_rollback_savepoints_free	释放 SAVEPOINT
trx_rollback_savepoint_free	
trx_release_savepoint_for_mysql	

### 10.3.9 XA

MySQL 中 InnoDB 对于 XA 功能的支持, 同事务的提交和回滚一样, 分为 MySQL Server 层和 InnoDB 层, 这两层都提供了对于 XA 的支持。整个 MySQL 系统对于 XA 功能的支持, 主要代码位于 MySQL 层, 即代码集中在 xa.cc 和 xa.h 文件当中。代码中包含了 XA 事务的 Start、Prepare、Commit、Rollback 等常规操作, 也包括了 XA 事务崩溃时的恢复操作, 如 trans\_xa\_start()、trans\_xa\_prepare()、trans\_xa\_commit()、trans\_xa\_rollback()、trans\_xa\_recover()、trans\_xa\_end() 等。

MySQL Server 对于 XA 事务状态的定义有别于 InnoDB 对于事务状态的定义, 内容为:

```
enum xa_states {XA_NOTR=0, XA_ACTIVE, XA_IDLE, XA_PREPARED, XA_ROLLBACK_ONLY};
```

InnoDB 对于 XA 的支持, 没有特定的提交和回滚操作, 这些操作和普通的单机事务没有区别, 所以共用了相同的代码。主要差异之处在于提供了一些特定函数诸如 trx\_commit\_or\_rollback\_prepare()、innobase\_xa\_prepare()、innobase\_xa\_recover() 等函数。

XA 属于分布式事务, 非本书重点, 期待有机会再进行介绍。

### 10.3.10 事务的其他内容

#### 1. 全局的事务对象

数据库引擎活动期间, 所有的事务包括用户发起的事务、系统自身产生的事务等信息以及相关的回滚段信息注册在结构体 trx\_sys\_t 中, 这是全系统的所有事务管理功能的注





册之处，如包括了 MVCC 管理、PURGE 清理工作的管理等。

```

/** The transaction system central memory data structure. */
struct trx_sys_t {...
    MVCC*          mvcc;          /*!< Multi version concurrency controlmanager */
    //MVCC机制的管理器，相关详情参见第12章
    volatile trx_id_t max_trx_id; /*!< The smallest number not yet assigned as
        a transaction id or transaction number. This is declared
                                volatile because it can be accessed without holding any mutex
                                during AC-NL-RO view creation. */
    //对于每个事务都有一个ID，已存在的所有事务的最大ID小于max_trx_id，这
    //表明max_trx_id是即将待分配的事务ID的最小值
    trx_ut_list_t   serialisation_list; //所有当前处于ACTIVE状态的读写事务按trx_
    t::no有序的一个列表
    ...
    trx_ut_list_t   rw_trx_list; //所有处于ACTIVE和COMMITTED状态的在内存的读写事务列
    表，按事务id倒排有序。Recovered事务和数据库引擎自身发起的事务在此列表
    ...
    trx_ut_list_t   mysql_trx_list; //所有的用户发起的事务列表。还包括正在发起的事务但还
    没有发起完成
    trx_ids_t       rw_trx_ids;    /*!< Read write transaction IDs */
    ...
    trx_rseg_t*     rseg_array[TRX_SYS_N_RSEGS]; //系统的回滚段，最大TRX_SYS_N_RSEGS为128个
    uint_t          rseg_history_len;
    trx_rseg_t*     const pending_purge_rseg_array[TRX_SYS_N_RSEGS];
    //用于PURGE操作的回滚段
    TrxIdSet        rw_trx_set;    /*!< Mapping from transaction id to transaction instance */
    uint_t          n_prepared_trx; /*!< Number of transactions currently in the
        XA PREPARED state */
    uint_t          n_prepared_recovered_trx; /*!< Number of transactions
        currently in XA PREPARED state that are also recovered.
                                Such transactions cannot be added during runtime. They can
                                only occur after recovery if mysqld crashed
                                while there were XA PREPARED transactions. We disable query
                                cache if such transactions exist. */
};

```

## 2. 事务 ID 的管理

在第二章曾经讨论过事务 ID 的作用，InnoDB 引擎需要使用事务 ID 区分事务，对于事务 ID 的管理方式有其生命周期。

首先，是事务 ID 的初始化，这是 InnoDB 引擎启动时需要为整个引擎完整准备环境（如各种数据结构初始化、缓存区准备、元数据加载等）的步骤之一，所以在引擎启动时从外存的加载上次系统生命周期内的事务最大 ID，然后增加出一段空余范围，生成新的事务 ID 供本次生命周期使用。这项工作是通过调用 `trx_sys_init_at_db_start()` 函数完成的。

```

/*****
Creates and initializes the central memory structures for the transaction
system. This is called when the database is started.
@return min binary heap of rsegs to purge */
purge_pq_t*
trx_sys_init_at_db_start(void) //从外存获得存储的事务ID, 生成新的事务ID, 新的事务ID总比
上次系统启动时候的事务ID大, 确保事务ID不重复
/*****
{...
    mtr_start(&mtr);
    sys_header = trx_sysf_get(&mtr); //从外存获得系统页的页头信息, 从这个页头上要找出存
    储的事务ID
    ...
    trx_sys->max_trx_id = 2 * TRX_SYS_TRX_ID_WRITE_MARGIN //增加出一段空余值
        + ut_uint64_align_up(mach_read_from_8(sys_header //从页头上要找出存储的事
        务ID, 然后增加出一段空余值, 作为新的事务ID
        + TRX_SYS_TRX_ID_STORE),
        TRX_SYS_TRX_ID_WRITE_MARGIN);
    ...
}

```

trx\_sys\_init\_at\_db\_start() 函数在系统启动期间被调用, 其调用关系如图 10-8 所示。

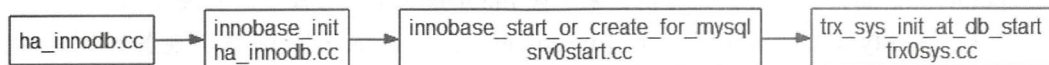


图 10-8 trx\_sys\_init\_at\_db\_start() 函数被调用的关系图

其次, 每个新事务需要获取新的事务 ID, 这是通过 trx\_sys\_get\_new\_trx\_id() 函数完成的。此函数多数情况下由 trx\_start\_low() 函数调用, 为每个新事务赋予新事务 ID 值。

```

trx_sys_get_new_trx_id()
{...
    if (!(trx_sys->max_trx_id % TRX_SYS_TRX_ID_WRITE_MARGIN)) {
        trx_sys_flush_max_trx_id(); //每256个事务ID赋值后, 往外存中保存一次, 避免较多IO, 避免系统
        宕机且下次启动后导致事务ID重复
    }
    return(trx_sys->max_trx_id++); //内存态的事务ID返回并自增
}

```

再次, 事务 ID 的作用, 主要如下:

- ❑ 作为标志写入日志: 如调用 row\_upd\_write\_sys\_vals\_to\_log() 写 RODO 日志。
- ❑ 标识对象生命状态与事务的关系: 如调用 dict\_build\_index\_def\_step() 标识索引是被本次生命周期 (系统从启动到停止为一个生命周期) 内的某个事务创建还是在其他什么周期内创建。
- ❑ 被用于检测某个事务的存活状态: 如调用 trx\_rw\_is\_active\_low()。



- ❑ 被用于 MVCC 技术中求解事务的快照：如调用 `ReadView::prepare(trx_id_t id)`。
  - ❑ 被用于统计信息中：如调用 `row_update_for_mysql_using_upd_graph()` 再调用 `srv_stats.n_rows_deleted.add((size_t)trx → id, 1)` 对某个事务的删除操作进行统计（所使用的“`srv_stats_tsrv_stats`”是内部的全局计数器）。
  - ❑ 被用于并发控制技术中：如调用 `row_unlock_for_mysql()` 函数判断是否元组是否可以解锁。
  - ❑ 被用于系统恢复中：如调用 `trx_resurrect_insert()` 从 UNDO 日志中恢复事务状态。
  - ❑ 被用于标识哪些事务处于活动状态：如调用 `trx_erase_from_write_set()` 从“`trx_sys → rw_trx_set`”中去除活动事务的注册信息。
- 最后，事务 ID 的其他特殊情况如下：
- ❑ 读写事务不赋予事务 ID（但是，需要写临时表的读写事务赋予事务 ID）。
  - ❑ PURGE 事务不赋予事务 ID。

## 10.4 InnoDB事务模型

在 InnoDB 层，InnoDB 的事务模型，实现了平板事务、链式事务<sup>①</sup>，并支持保存点（Savepoint）。下面我们通过代码分析，来掌握 InnoDB 对于事务模型的实现技术。

首先，在 MySQL Server 层，通过 `mysql_execute_command()` 函数实现各种用户 SQL 的命令分发，这样当接收到事务相关的命令时，就执行相应的事务处理功能，如开始事务、提交事务、回滚事务等。如下面的代码分析中的“`case SQLCOM_BEGIN`”等语句。

其次，开始事务、提交事务、回滚事务这三者，组成了平板事务，在这样的一个事务块内部，不存在层次，事务块的语句要么都提交要么都回滚。因此，正常的平板事务是很容易实现的。如下代码分析中的“`case SQLCOM_COMMIT`”和“`case SQLCOM_ROLLBACK`”语句中如果去掉对“`tx_chain`”的判断的相关代码，则是平板事务的实现方式。

再次，当在如下代码分析中的“`case SQLCOM_COMMIT`”和“`case SQLCOM_ROLLBACK`”语句中如果包括对“`tx_chain`”的判断的相关代码，则是链式事务的实现方式。在对“`tx_chain`”的判断的相关代码中，又调用 `trans_begin()` 函数自动开始一个事务，且对会话中的事务隔离级别做了继承为上一个事务的隔离级别值（不重新设置）或重新设置为默认值的处理，这样就实现了链式事务的语义。

最后，使用 `SAVEPOINT`、`ROLLBACK [WORK] TO [SAVEPOINT] identifier`、`RELEASE`

① MySQL官方手册：The AND CHAIN clause causes a new transaction to begin as soon as the current one ends, and the new transaction has the same isolation level as the just-terminated transaction. The RELEASE clause causes the server to disconnect the current client session after terminating the current transaction. Including the NO keyword suppresses CHAIN or RELEASE completion, which can be useful if the completion\_type system variable is set to cause chaining or release completion by default.

SAVEPOINT identifier 等命令支持 SAVEPOINT 技术，意味着在如下代码中的“case SQLCOM\_SAVEPOINT”等判断语句配合“case SQLCOM\_BEGIN”“case SQLCOM\_COMMIT”和“case SQLCOM\_ROLLBACK”实现平板事务对 SAVEPOINT 的支持。而 SAVEPOINT 的存在，使得事务有了“层次”。但这种层次不是事务嵌套的嵌套造成的层次，而是一种线状的层次，可被跨越而直接回滚到其中某个层次上。所以 MySQL 以及 InnoDB 不支持嵌套事务模型。

```
mysql_execute_command(THD *thd) //MySQL Server层的命令分发器
{...
switch (lex->sql_command) {
    case SQLCOM_BEGIN: //用户使用BEGIN、START TRANSACTION命令显式开始事务
        if (trans_begin(thd, lex->start_transaction_opt))
            //参见10.3.2节，逐层调用到InnoDB里面，开始事务
        ...
    case SQLCOM_COMMIT: //用户使用COMMIT命令显式提交事务
        {...
            bool tx_chain= (lex->tx_chain == TVL_YES ||
                //用户使用COMMIT命令带有“AND [NO] CHAIN”关键字，确定是否要进行链式事务
                (thd->variables.completion_type == 1 &&
                    lex->tx_chain != TVL_NO));

            ...

            if (trans_commit(thd)) //参见10.3.3节，逐层调用到InnoDB里面，提交事务
                goto error;
            thd->mdl_context.release_transactional_locks();
            /* Begin transaction with the same isolation level. */
            if (tx_chain) //如果是链式事务，开始新事务，但事务的隔离级别是继承自上一个事务的
            {
                if (trans_begin(thd)) //开始新事务，隔离级别是继承自上一个事务
                    goto error;
            }
            else //如果不是链式事务，修改会话（session）的隔离级别为系统配置参数指定的默认值，
                //但没有调用trans_begin(thd)开始新事物
            {
                /* Reset the isolation level and access mode if no chaining transaction.*/
                thd->tx_isolation= (enum_tx_isolation) thd->variables.tx_isolation;
                //修改会话（session）的隔离级别为系统配置参数指定的默认值
                thd->tx_read_only= thd->variables.tx_read_only;
            }
            ...
        }
    case SQLCOM_ROLLBACK: //参见10.3.5节，逐层调用到InnoDB里面，回滚事务
        {...
            bool tx_chain= (lex->tx_chain == TVL_YES ||
                //用户使用ROLLBACK命令带有“AND [NO] CHAIN”关键字，确定是否要进行链式事务
                (thd->variables.completion_type == 1 &&
```



```

lex->tx_chain != TVL_NO));

...

if (trans_rollback(thd)) //回滚新事务，隔离级别是继承自上一个事务
    goto error;
thd->mdl_context.release_transactional_locks();
/* Begin transaction with the same isolation level. */
if (tx_chain) //如果是链式事务，开始新事务，但事务的隔离级别是继承自上一个事务的
{
    if (trans_begin(thd))
        goto error;
}
else //如果不是链式事务，修改会话（session）的隔离级别为系统配置参数指定的默认值，
    但没有调用trans_begin(thd)开始新事物
{
    /* Reset the isolation level and access mode if no chaining transaction.*/
    thd->tx_isolation= (enum_tx_isolation) thd->variables.tx_isolation;
    //修改会话（session）的隔离级别为系统配置参数指定的默认值
    thd->tx_read_only= thd->variables.tx_read_only;
}

...
}
case SQLCOM_RELEASE_SAVEPOINT: //用户使用RELEASE SAVEPOINT命令显式释放SAVEPOINT
    if (trans_release_savepoint(thd, lex->ident))
...
case SQLCOM_ROLLBACK_TO_SAVEPOINT: //用户使用ROLLBACK [WORK] TO [SAVEPOINT]
命令显式回滚SAVEPOINT
    if (trans_rollback_to_savepoint(thd, lex->ident))
...
case SQLCOM_SAVEPOINT: //用户使用SAVEPOINT命令显式命名SAVEPOINT
    if (trans_savepoint(thd, lex->ident))
...
...}
...}

```

第五，如果使用 BEGIN、COMMIT 等语句，是显式的开始一个事务。而数据库引擎通常支持自动提交，支持语句级事务，这意味着某些操作可以直接启动事务，这些不是显式启动的事务，被称为隐式事务。InnoDB 对隐式事务的支持，就不是通过上面的代码进行的，而是把事务的开启嵌入在了不同的命令当中。一个隐式事务的开启的一个调用栈如下（INSERT 语句事例）：

```

trx_start_low(trx_t * trx, bool read_write) Line 1367C++ //InnoDB内部开始事务
trx_start_if_not_started_xa_low(trx_t * trx, bool read_write) Line 3102C++
//在InnoDB内部隐式启动一个事务
row_insert_for_mysql_using_ins_graph(const unsigned char * mysql_rec, row_
prebuilt_t * prebuilt) Line 1675C++
row_insert_for_mysql(const unsigned char * mysql_rec, row_prebuilt_t * prebuilt)

```

```

Line 1819 C++
ha_innbase::write_row(unsigned char * record) Line 7289C++
handler::ha_write_row(unsigned char * buf) Line 7753C++
write_record(THD * thd, TABLE * table, COPY_INFO * info, COPY_INFO * update) Line 1858C++
Sql_cmd_insert::mysql_insert(THD * thd, TABLE_LIST * table_list) Line 780C++
//执行INSERT语句
Sql_cmd_insert::execute(THD * thd) Line 3124C++
mysql_execute_command(THD * thd, bool first_level) Line 3319C++

```

之后，隐式事务的提交调用栈如下：

```

trx_commit_low(trx_t * trx, mtr_t * mtr) Line 2195C++ //InnoDB中的事务提交
trx_commit(trx_t * trx) Line 2239C++
trx_commit_for_mysql(trx_t * trx) Line 2469C++
innbase_commit_low(trx_t * trx) Line 3938C++
innbase_commit(handlerton * hton, THD * thd, bool commit_trx) Line 4097C++
ha_commit_low(THD * thd, bool all, bool run_after_commit) Line 1754C++
TC_LOG_DUMMY::commit(THD * thd, bool all) Line 30C++
ha_commit_trans(THD * thd, bool all, bool ignore_global_read_lock) Line 1649C++
trans_commit_stmt(THD * thd) Line 460C++
mysql_execute_command(THD * thd, bool first_level) Line 4747C++
//调用了trans_commit_stmt()

```

隐式事务被提交或回滚的相关代码如下：

```

mysql_execute_command(THD *thd) //MySQL Server层的命令分发器
{...
finish: //mysql_execute_command()函数的函数尾部分
...
    if (! thd->in_sub_stmt) //不是多语句事务
    {...
        if (thd->is_error() || (thd->variables.option_bits & OPTION_MASTER_SQL_ERROR))
trans_rollback_stmt(thd); //如果有错误发生，必须回滚单语句事务，调用ha_rollback_
                                trans()完成回滚
        else
        {
            /* If commit fails, we should be able to reset the OK status. */
            thd->get_stmt_da()->set_overwrite_status(true);
trans_commit_stmt(thd); //否则，提交单语句事务
            thd->get_stmt_da()->set_overwrite_status(false);
        }
    }
...
    if (! thd->in_sub_stmt && thd->transaction_rollback_request)
    {
        /*
            We are not in sub-statement and transaction rollback was requested by

```



```

        one of storage engines (e.g. due to deadlock). Rollback transaction in
        all storage engines including binary log.
    */
    trans_rollback_implicit(thd); //调用ha_rollback_trans() 隐式回滚事务
    thd->mdl_context.release_transactional_locks();
}
else if (stmt_causes_implicit_commit(thd, CF_IMPLICIT_COMMIT_END))
{...
    /* Commit the normal transaction if one is active. */
    trans_commit_implicit(thd); //调用ha_commit_trans() 隐式提交事务
    thd->get_stmt_da()->set_overwrite_status(false);
    thd->mdl_context.release_transactional_locks();
}
...
}

```

最后，假设实现嵌套事务，又该怎么做呢？

在 10.2.1 节，讲述事务状态的时候，我们特别提醒：事务模型的（平板事务、链式事务、嵌套事务等）实现，严重依赖于事务的状态以及状态变迁，所以事务的状态标识是事务模型实现的一个重要元素。

如果支持嵌套事务，就需要增加相应的事务状态，以对事务所处的阶段进行识别，比如说，事务处于 ACTIVE 状态，还可以开始新的子事务，则需要增加子事务的 ACTIVE、COMMITTED、ROLLBACKED 等状态，一个事务中有多个子事务则需要对多个子事务分别做状态区分。总之，实现事务管理就是实现一个事务状态驱动机，实现嵌套事务就是在简单的事务状态驱动机上增加复杂的事务状态变迁。

## 10.5 本章小结

本章分析了 InnoDB 引擎在事务管理方面的主要内容，第一节是一个概述，从整体和文件结构上介绍了 InnoDB 的事务和并发控制的相关内容；第二节与第三节介绍了事务的数据结构和代码流程，对事务管理的框架结构、主要功能点、事务模型等做了深入剖析；第四节基于前面的内容，总结了 InnoDB 提供的事务模型和事务模型的实现技术；期望这些内容能够对读者掌握事务模型的处理技术有帮助。

# InnoDB 并发控制系统的实现——两阶段锁

在 InnoDB 内部，锁有两大类型，一种是系统锁，一种是事务锁，在 PostgreSQL 中，系统锁属于 latch 锁类型，事务锁属于 lock 锁类型，但 InnoDB 没有做如此区分：

- 系统锁：是保护共享的内存数据结构，不被并发的 SESSION 同时修改。如各种 MUTEX（10.3.10 节提及的 `trx_sys` → `mutex`、事务锁管理需要使用的 `lock_sys` → `mutex`、回滚段内存结构管理需使用的 `rseg` → `mutex` 等）。
- 事务锁：用户 SQL 执行过程中，保护用户表中的数据不被并发的 SESSION 同时修改，如我们常提及的读锁、写锁、意向锁。事务锁是事务并发访问控制技术中基于锁的并发控制技术所指的锁。InnoDB 事务锁的操作，符合两阶段提交的规则。第 11.3 节事务锁的讨论，是本章的重点。

## 11.1 锁的概述

对于锁的理解，首先不应局限于事务锁的范围，而是纵观整个数据库系统，查看所有与“锁定”操作相关的内容，先明了锁的本质，然后再根据数据库管理系统的业务需求，对锁归类区分。

### 11.1.1 锁操作的本质

在第 7 章，讲述 PostgreSQL 的锁操作的本质时，描述如下：

锁操作，所起的作用就是防止被锁的对象被并发操作同时修改。从技术本质上看，加锁操作就是为特定对象设置一个标志位，然后通过使用锁的机制（对象上存在标志位则不能



改写，放弃加锁请求或等待锁释放后再进行操作）和释放锁（取消特定对象上被设置的标志位），一共三个过程，共同完成排斥其他并发操作对这个特定对象进行操作。这是在说，锁的本质，是保护共享资源不被并发修改破坏（原子操作不需要锁）。

锁的另外一层含义，是抑制并发。保护共享资源即在抑制并发，这只是抑制并发的一部分。而抑制并发的另外一部分含义，常用在多个锁同时施加的情况，如 ABBA 方式（严格要求配对使用，消除死锁隐患）。存在以下情况：

```
Lock(A_lock);    //抑制想要获取A_lock锁的并发
...
Lock(B_lock);    //主要目的是保护共享资源，同时抑制想要获取B_lock锁的并发
shared_resource = update...;
Unlock(B_lock);  //必须配对使用
...
Unlock(A_lock);
```

InnoDB 的锁操作的本质，同上面的描述。

### 11.1.2 全局锁表

MySQL 整个数据库引擎，提供了两种类型的全局锁表，一是元数据锁表，二是行级锁表。

MySQL Server 层提供了一个全局的元数据锁表“MDL\_map m\_locks”（参见 11.4.1 节的“8. 元数据锁的集合”），用以处理 DDL 之间的并发、处理 DDL 和 DML 之间的并发，并发冲突的根源在于对于元数据的竞争。

InnoDB 提供了全局的行级锁表（lock\_sys\_t），用以记录 InnoDB 系统运行期间所有行级锁的施加情况，这应对的是对于数据操作的 DML 与 DQL 之间的并发，并发冲突的根源在于对于表中数据的竞争。行级锁包括记录锁、谓词锁、谓词页锁，各种类型的锁，在全局行级锁表中存放到了不同的 Hash 表里面（三种锁对应的三个 Hash 表的介绍参见 11.7 节）。

### 11.1.3 封锁系统的架构

在详细讨论封锁技术和 MVCC 技术的实现之前，我们先从并发控制系统的架构，整体上认识一下 MySQL 以及 InnoDB 的并发控制系统。

说明：

- 首先，图 11-1 分为上下两个部分，上面的部分，是 MySQL Server 层，这一层主要完成元数据锁的加锁、解锁以及死锁检测工作。
- 其次，图 11-1 下面的部分，是 InnoDB 层，这一层主要完成行级锁的加锁、解锁以及死锁检测工作。所以，元数据锁和行级锁各自处理自己级别的死锁。





- MDL\_context: 每个会话即用户连接（一个物理的线程）上存在一个元数据锁的上下文，用以表示本会话生命期内（实则是本会话的当前事务生命期内）的各种锁的施加、获得等情况。
- MDL\_ticket: 表示接收到加锁申请，加锁者买到一张入场券，但需要准备接受安检。安检的结果是：一个加锁请求即 MDL\_request 对象来临后，锁可以被授予也可以被拒绝（被拒绝则发出加锁请求的会话处于等待故设置其 MDL\_context 上的“m\_wait”）。如果被授予，则生成一个 MDL\_lock 对象，绑定在 MDL\_ticket 对象上，此时加锁请求成功完成。
- 最后，图 11-1 中的带有圆圈的数字 1 到 4，表明了上述逻辑的对应关系。如带有圆圈的数字 1 表明：MDL\_lock 对象全部要注册到全局的 Hash 表“m\_locks”里，以便进行锁的查找。

## 11.2 系统锁

InnoDB 提供了两种系统锁，用于多线程间同步与互斥。第一种是读写锁，使用读锁和写锁实现互斥并发的会话对同一个内存对象（如数据缓存区）的修改操作，包括用户发起的会话和系统内部发起的会话引发的并发操作。第二种是 Mutex 锁，利用操作系统的 Mutex 功能对共享对象加锁，以完全互斥的方式保护内存中的数据结构（如全局锁表）。

InnoDB 提供的读写锁和 Mutex，都相当于 PostgreSQL 的 Latch。

### 11.2.1 读写锁

InnoDB 提供一种自旋锁，是基于操作系统的 Test-And-Set 原子指令实现，在 InnoDB 内部被称为读写锁（read-write lock）。

#### 1. 数据结构

读写锁，使用一个结构体定义，这个结构体定义了读写锁包括的成员内容。

```
/** The structure used in the spin lock implementation of a read-write
lock. Several threads may have a shared lock simultaneously in this
//读锁可以有多个施加者
lock, but only one writer may have an exclusive lock, in which case no
//写锁只有一个施加者
shared locks are allowed. To prevent starving of a writer blocked by
readers, a writer may queue for x-lock by decrementing lock_word: no
//排它锁/写锁，写操作施加x锁
new readers will be let in while the thread waits for readers to
exit. */
struct rw_lock_t
```

```

#ifdef UNIV_SYNC_DEBUG
    : public latch_t
#endif /* UNIV_SYNC_DEBUG */
{
    volatile lintlock_word; //锁的值，真正的记录锁的状态标志变量，读写锁就是在此设置不同的值
    表示不同的锁
    volatile ulint waiters; /*!< 1: there are waiters */ //谁在等待本锁
    volatile ibool recursive; //缺省值为FALSE，不递归。
    volatile ulint sx_recursive; /*!< number of granted SX locks. */
    volatile os_thread_id_t writer_thread; //写操作的线程的ID
    os_event_t event; /*!< Used by sync0arr.cc for thread queueing */ //OS事件
    os_event_t wait_ex_event; //下一个写操作等待者正在等待的OS事件
    /*!< Event for next-writer to wait on. A thread must decrement lock_word
    before waiting. */
#ifdef INNODB_RW_LOCKS_USE_ATOMICS
    mutable ib_mutex_tmutex; /*!< The mutex protecting rw_lock_t */
    //Mutex锁，ib_mutex_t的描述参见11.2.2节
#endif /* INNODB_RW_LOCKS_USE_ATOMICS */

    UT_LIST_NODE_T(rw_lock_t) list; //所有的读写锁list
    ...
};

```

## 2. 读写锁的粒度

InnoDB 的读写锁，定义了三种粒度，分别如下：

```

enum rw_lock_type_t {
    RW_S_LATCH = 1, //共享锁
    RW_X_LATCH = 2, //排它锁
    RW_SX_LATCH = 4, //意向排它锁，阻塞写操作，不阻塞读操作
    RW_NO_LATCH = 8 //没有锁
};

```

## 3. 读写锁的创建和释放

读写锁在某个子系统被初始化的时候被创建，如日志初始化、数据字典初始化的时候。

```

rw_lock_create_func( //在一个特定的内存位置中创建或初始化一个读写锁对象
rw_lock_t* lock, /*!< in: pointer to memory */ //传入参数，指向一个读写锁
...
{...
    lock->lock_word = X_LOCK_DECR; //define X_LOCK_DECR 0x20000000。给锁赋予一个初值
    lock->waiters = 0;
    ...
    lock->event = os_event_create(0); //为一些成员赋值
    lock->wait_ex_event = os_event_create(0); //为一些成员赋值
}

```



```

mutex_enter(&rw_lock_list_mutex); //使用mutex保护rw_lock_list
ut_ad(UT_LIST_GET_FIRST(rw_lock_list) == NULL
    || UT_LIST_GET_FIRST(rw_lock_list)->magic_n == RW_LOCK_MAGIC_N);
UT_LIST_ADD_FIRST(rw_lock_list, lock);
mutex_exit(&rw_lock_list_mutex);
}

```

读写锁在被创建的时候,是通过一个 `rw_lock_create` 宏完成 `rw_lock_create_func()` 函数调用的。

```

rw_lock_create( //rw_lock_create宏调用了rw_lock_create_func()函数
checkpoint_lock_key, &log_sys->checkpoint_lock, //一个key "checkpoint_lock_key"
对应着一个读写锁 "checkpoint_lock"
SYNC_NO_ORDER_CHECK);

```

读写锁被 `rw_lock_free_func()` 函数释放,多数情况下是锁不再被使用的时候才释放读写锁,如数据 buffer 在系统停止前被关闭。函数实现和上下文较为简单,不再赘述。

#### 4. 加锁

加锁,是在锁结构体上的 `lock_word` 成员上减去一些固定数值,然后用新的 `lock_word` 值与一个边界值做判断,确认锁的状态(锁的状态可以参见表 11-1)。

##### (1) 一加共享锁

对于读写锁,加共享锁操作是通过如下调用栈完成的。

```

rw_lock_s_lock(M)
->rw_lock_s_lock_func()
->rw_lock_s_lock_low()
->rw_lock_lock_word_decr()

```

首先是 `rw_lock_s_lock_func()` 函数尝试加锁。

```

rw_lock_s_lock_func(
    rw_lock_t* lock, /*!< in: pointer to rw-lock */
    ...
{...
    if (!rw_lock_s_lock_low(lock, pass, file_name, line)) {
        //如果加锁不成功,进入等待状态
        /* Did not succeed, try spin wait */
        rw_lock_s_lock_spin(lock, pass, file_name, line);
        //多种机制继续尝试加锁。重要函数详见11.2.1.7节
    }
}

```

其次,通过 `rw_lock_s_lock_low()` 函数为再下层的函数的参数传递合适的值。注意不同的加锁粒度,以及参数的值是不同的。

```

ibool

```

```

rw_lock_s_lock_low(
    rw_lock_t*    lock,    /*!< in: pointer to rw-lock */
    ...
{
    if (!rw_lock_lock_word_decr(lock, 1, 0)) {
        //注意这里的参数和下面对这些参数值对加锁的影响分析
        /* Locking did not succeed */
        return(FALSE);
    }
    ...
    return(TRUE);    /* locking succeeded */
}

```

最后，调用 `rw_lock_lock_word_decr()` 函数完成真正的加锁操作。

```

bool
rw_lock_lock_word_decr(    //加锁操作的底层函数
    rw_lock_t*    lock,    /*!< in/out: rw-lock */
    uint          amount,    /*!< in: amount to decrement */
    //在原来的锁值上准备减少的数
    lint          threshold) /*!< in: threshold of judgement */
    //某种锁的边界值。上层rw_lock_s_lock_low()函数传入的threshold值为0
{
    #ifdef INNODB_RW_LOCKS_USE_ATOMICS    //如果读写锁被定义为原子操作，定义了这个宏，加锁操作
    将使用无锁化(latch-free)的编程技术，使得加锁操作更轻量，推荐在提供了数__sync_bool_
    compare_and_swap函数的操作系统上使用无锁化(latch-free)的编程
    ...
    local_lock_word = lock->lock_word;    //保存一份原值，为实现无锁化编程做好准备
    while (local_lock_word > threshold) {    //条件为真，满足加锁条件。threshold值为0，
    表示local_lock_word的值大于0，即可以加锁
        if (os_compare_and_swap_lint(&lock->lock_word, //os_compare_and_swap_lint
            local_lock_word,
            local_lock_word - amount)) {    //加锁的操作，是在锁标志位上减去一个数，这里是用减去数后的值
            替换原值
                return(true);
            }
            local_lock_word = lock->lock_word;
        }
        return(false);
    #else /* INNODB_RW_LOCKS_USE_ATOMICS */    //否则，锁是非原子的
        bool success = false;
        mutex_enter(&(lock->mutex));
        if (lock->lock_word > threshold) {    //条件为真，满足加锁条件。threshold值为0，表示
            local_lock_word的值大于0，即可以加锁

```

⊖ 无锁化编程，latch-free/lock-free，效率高，如欲进阶可自行查阅相关资料。



```
lock->lock_word -= amount; //加锁的操作，是在锁标志位上减去一个数
    success = true;
}
mutex_exit(&(lock->mutex));
return(success);
#endif /* INNODB_RW_LOCKS_USE_ATOMICS */
}
```

表 11-1 锁值与锁的状态关系

条件	说明
lock_word==X_LOCK_DECR	不存在任何锁
HALF_DECR < lock_word < DECR	施加了 S 锁，没有写者在等待锁
lock_word == HALF_DECR	施加了 SX 锁，没有写者在等待锁
0 < lock_word < HALF_DECR	施加了 S 和 SX 锁，没有写者在等待锁
lock_word == 0	施加了 X 锁，没有写者在等待锁
- HALF_DECR < lock_word < 0	施加了 S 锁，有写者在等待锁
lock_word == - HALF_DECR	施加了 X 和 SX 锁，没有写者在等待锁
- DECR < lock_word < - HALF_DECR	施加了 S 锁，拥有意向锁的写者在等待锁
lock_word == - DECR	施加了 X 锁，且存在递归的 X 锁申请（2 个 X 锁）
-(DECR + HALF_DECR) < lock_word < - DECR	施加了 X 锁，有多个 X 锁其个数为 2 - (lock_word + DECR)
lock_word == -(DECR + HALF_DECR)	施加了 X 和 SX 锁，且存在递归的 X 锁申请（2 个 X 锁）
lock_word < -( DECR + HALF_DECR)	施加了 X 和 SX 锁，有多个 X 锁其个数为 2 - (lock_word + DECR + HALF_DECR)

说明：

❑ 使用 HALF\_DECR 表示 X\_LOCK\_HALF\_DECR。

❑ 使用 DECR 表示 X\_LOCK\_DECR。

(2) 加排它锁或意向锁

InnoDB 调用 rw\_lock\_x\_lock\_low() 施加排它锁，调用 rw\_lock\_sx\_lock\_low() 施加意向锁，施加锁的代码都要调用 rw\_lock\_lock\_word\_decr() 函数，加锁的代码较施加读锁略为复杂一些，但也相对简单，本节不再赘述，相关信息，可以参见表 11-2 和源码。

表 11-2 各种锁的加锁判断条件

加锁粒度	amount 值	Threshold 值	说明：加锁成功的条件和对应操作 if (lock->lock_word > threshold) lock->lock_word -= amount;
S 锁	1	0	如锁的初始值为 X_LOCK_DECR，必定大于 0，可以多次减去 1，意味着读锁可以被施加多次
X 锁	X_LOCK_DECR	X_LOCK_HALF_DECR	如锁的初始值为 X_LOCK_DECR，必定大于 X_LOCK_HALF_DECR，可以减去 X_LOCK_DECR 值 1 次，这样锁上的值就不能大于 X_LOCK_HALF_DECR，意味着写锁只能被施加 1 次
	X_LOCK_DECR	0	

(续)

加锁粒度	amount 值	Threshold 值	说明：加锁成功的条件和对应操作 if (lock->lock_word > threshold) lock->lock_word -= amount;
SX 锁	X_LOCK_HALF_DECR	X_LOCK_HALF_DECR	<p>如锁的初始值为 X_LOCK_DECR, 必定大于 X_LOCK_HALF_DECR, 可以减去 X_LOCK_DECR 值 1 次, 这样锁上的值就不能大于 X_LOCK_HALF_DECR, 意味着意向锁只能被施加 1 次。</p> <p>之后, 如果施加读锁, 则意味锁标志位上的值大于 0, 可以继续施加读锁, 即意向锁不排斥读锁。</p> <p>但是, 一旦施加了意向锁之后, 据表头的判断条件, 写锁是不可能施加的, 即意向锁排斥写锁</p>

但是, 对于锁标识 lock\_word 的值, 对其修改, 不只是加锁的 rw\_lock\_lock\_word\_decr() 函数和解锁的 rw\_lock\_lock\_word\_incr() 函数, 而是诸如 rw\_lock\_x\_lock\_func\_nowait() 函数也可以修改, 所以表 11-1 中的“说明”列的信息不能完全表明锁标识 lock\_word 的值与锁的对应关系, 而一个正确的对应关系, 应该如表 11-3 所示。

表 11-3 读写锁相容

		Granted Mode, 已经授予的锁		
Requested Mode 正申请的锁		S	SX	X
	S	Y	Y	
	SX	Y		
	X			

说明：“Y”表示可以被授予锁，空白则表示不可以授予新请求的锁。

### 5. 解锁

解锁的操作方式, 与加锁的操作方式正好相反, 解锁需要对 lock\_word 的值进行加数操作, 加数后的新 lock\_word 值表明解锁后的状态, 新状态可以参见表 11-4。

各种锁的解锁操作, 需要调用 rw\_lock\_lock\_word\_incr() 函数完成, 只是增加数值的参数值不同, 如表 11-1 所示。

表 11-4 解锁操作所加的数值情况

解锁粒度	所加的数值	说明
S	1	读锁可以一个一个地通过数值逐一递增而解锁
SX	X_LOCK_HALF_DECR	意向锁可以通过数值逐半增加而解锁
X	X_LOCK_DECR	写锁可以通过数值一次增为初始值而解锁 (非多个写锁解锁)

### 6. 读写锁的作用

读写锁作为系统锁被用到了系统级别的共享资源互斥上, 如数据缓存区中数据页的获取。



```

buf_page_optimistic_get( //数据缓存区中数据页的获取
{...
    switch (rw_latch) {
    case RW_S_LATCH:
        success = rw_lock_s_lock_nowait(&block->lock, file, line); //施加读写锁的共享锁
        fix_type = MTR_MEMO_PAGE_S_FIX;
        break;
    case RW_X_LATCH:
        success = rw_lock_x_lock_func_nowait_inline(&block->lock, file, line);
        //施加读写锁的排它锁
        fix_type = MTR_MEMO_PAGE_X_FIX;
        break;
    default:
        ut_error; /* RW_SX_LATCH is not implemented yet */
    }
    ...
}

```

如在系统内部的字典表里保存元信息，也需要使用读写锁。

```

dict_stats_save( //字典表信息保存
    dict_table_t*table_orig,
    const index_id_t*only_for_index)
{...
    rw_lock_x_lock(&dict_operation_lock); //施加读写锁的排它锁
    ...
    rw_lock_x_unlock(&dict_operation_lock); //释放读写锁的排它锁
    ...
}

```

第三种用法，是在读写锁的基础上，为其他对象加锁和解锁进行封装。如在内存 Hash 表上的封装。

```

hash_lock_x( //Hash表加锁
    hash_table_t* table, /*!< in: hash table */
    ulint fold) /*!< in: fold */
{
    rw_lock_t* lock = hash_get_lock(table, fold);
    ...
    rw_lock_x_lock(lock); //Hash表加锁
    ...
}

hash_unlock_x( //Hash表解锁
    hash_table_t* table, /*!< in: hash table */
    ulint fold) /*!< in: fold */
{

```

```

    rw_lock_t* lock = hash_get_lock(table, fold);
...
    rw_lock_x_unlock(lock); //Hash表解锁
}

```

## 7. rw\_lock\_s\_lock\_spin() 函数分析

如果加读锁不成功，则需要调用本函数，尝试继续加锁。但是，尝试加锁的机制可单一可丰富，本函数就是一个丰富的、有多种尝试方式的继续加锁机制的函数。

```

/*****
Lock an rw-lock in shared mode for the current thread. If the rw-lock is
locked in exclusive mode, or there is an exclusive lock request waiting,
the function spins a preset time (controlled by srv_n_spin_wait_rounds), waiting
for the lock, before suspending the thread. */
void
rw_lock_s_lock_spin( //多次尝试加锁，尝试多次加锁的机制比较丰富，值得多研究本函数体的内容
    rw_lock_t* lock, /*!< in: pointer to rw-lock */
    ulint pass, /*!< in: pass value; != 0, if the lock will be passed to
another thread to unlock */
    const char* file_name, /*!< in: file name where lock requested */
    ulint line) /*!< in: line where requested */
{
    ulint i = 0; /* spin round count */
    sync_array_t* sync_arr; //同步数组，包含有等待队列
...
lock_loop:
    /* Spin waiting for the writer field to become free */
    os_rmb;
    while (i < srv_n_spin_wait_rounds⊖ && lock->lock_word <= 0) {
        //没有超出轮询次数 且 有写锁存在，则延迟会一会后继续探测
        if (srv_spin_wait_delay⊖) {
            ut_delay(ut_rnd_interval(0, srv_spin_wait_delay)); //延迟会一会
        }
        i++;
    }

    if (i >= srv_n_spin_wait_rounds) { //超出轮询次数，使得本线程睡眠
        os_thread_yield(); //放弃自己（本线程）的时间片
    }

    /* We try once again to obtain the lock */
    if (TRUE == rw_lock_s_lock_low(lock, pass, file_name, line)) { //再尝试一次加锁
        rw_lock_stats.rw_s_spin_round_count.inc();
    }
}

```

⊖ 对应innodb\_sync\_spin\_loops参数，控制spin lock的轮询次数，默认值为30，可动态调整。

⊖ 对应innodb\_spin\_wait\_delay参数，控制spin lock的轮询间隔，默认值为6，可动态调整。



```

        return; /* Success */
    } else {
        if (i < srv_n_spin_wait_rounds) { //没有超出轮询次数,则重新尝试执行本函数主体代码
            goto lock_loop;
        }
        rw_lock_stats.rw_s_spin_round_count.inc();
        sync_cell_t* cell;
        sync_arr = sync_array_get_and_reserve_cell(lock, RW_LOCK_S, file_name,
            line, &cell); //因不能获得锁,把自己放入等待队列

        /* Set waiters before checking lock_word to ensure wake-upsignal is sent.
           This may lead to some unnecessary signals. */
        rw_lock_set_waiter_flag(lock); //设置自己为等待者,直到自己被唤醒

        if (TRUE == rw_lock_s_lock_low(lock, pass, file_name, line)) {
            //如果能够获得锁,则把自己从等待队列中释放出来
            sync_array_free_cell(sync_arr, cell);
            //把自己从等待队列中释放出来
            return; /* Success */
        }

        ...

        sync_array_wait_event(sync_arr, cell);
        //如果不能够获得锁,产生一个等待事件,并判断是否有死锁发生
        i = 0;
        goto lock_loop;
    }
}

```

## 11.2.2 Mutex 锁

InnoDB 使用 Mutex 保护一些基本的结构体等对象,但没有直接使用操作系统定义好的 Mutex,而是自定义了 Mutex。本节将从多个角度,来探讨 InnoDB 对于 Mutex 的多个方面的内容。

### 1. InnoDB 定义的 Mutex

InnoDB 在 `ib0mutex.h` 文件中定义了六种自定义的 Mutex<sup>①</sup>:

- ❑ **OSBasicMutex**: OS 的 Event 定义的 Mutex,依赖于具体的 OS;施加锁只尝试一次。如下面的初始化代码段等。
- ❑ **OSTrackMutex**: 继承自 OSBasicMutex,依赖于 OS;施加锁尝试函数参数指定的次数 (`max_spins` 参数指定尝试的次数)。
- ❑ **TTASFutexMutex**: 利用 OS 的 CAS 原子指令实现锁的施加尝试,利用 TAS 实现解锁尝试。适用于 Linux 的 Futex 机制<sup>②</sup>。

① 自定义多种的Mutex起因: <http://dev.mysql.com/worklog/task/?id=6044>。

② Futex是Fast Userspace muTexes的缩写,被Linux 2.5.7支持。Futex是用户态与内核态混合的同步机制,同步的进程间通过mmap共享一段内存, futex变量位于这段共享的内存中且操作是原子的,当进程尝试进入互斥区或者退出互斥区的时候,先查看共享内存中的futex变量,如果没有竞争发生,则只修改futex,而不用再执行系统调用,这样会比没有Futex机制总是需要进入内核态做判断更高效。

❑ **TTASMutex**：利用 OS 的 TAS 原子指令实现锁的施加和解锁尝试，在尝试不成的情况下，进行忙等待。

❑ **TTASEventMutex**：利用 OS 的 TAS 原子指令实现锁的施加和解锁尝试，在尝试不成的情况下，每次尝试之间需要调用 `ut_delay()` 进行睡眠延迟，尝试指定的次数（`max_spins` 参数指定尝试的次数），且最终还要等到锁为止。

❑ **PolicyMutex**：获得锁，且获得施加锁者，主要用于 Performance schema 监控。

**OSBasicMutex** 的初始化 `init()` 函数的代码段需要区分操作系统，相应的 `enter()`、`destroy()`、`exit()` 等函数也需要区分操作系统，根据不同的操作系统调用不同操作系统的相应函数：

```
#ifdef _WIN32
InitializeCriticalSection((LPCRITICAL_SECTION) &m_mutex);
//Mutex的定义依赖于Windows的InitializeCriticalSection()函数
#else
{
    int    ret;
    ret = pthread_mutex_init(&m_mutex, MY_MUTEX_INIT_FAST);
    //Mutex的定义依赖于Posix Thread的pthread_mutex_init()函数
    ut_a(ret == 0);
}
#endif /* _WIN32 */
```

**OSBasicMutex** 的测试锁是否施加 `try_lock()` 函数的代码段如下：

```
/** @return true if locking succeeded */
bool try_lock() UNIV_NOTHROW
{
    ut_ad(innodb_calling_exit || !m_freed);
#ifdef _WIN32
    return(TryEnterCriticalSection(&m_mutex) != 0);
    //Mutex的trylock依赖于Windows函数
#else
    return(pthread_mutex_trylock(&m_mutex) == 0);
    //Mutex的trylock依赖于Posix Thread
#endif /* _WIN32 */
}
```

TAS 和 CAS 原子操作的定义如下：

```
#ifdef _WIN32
# define TAS(l, n) os_atomic_test_and_set_u32((l), (n))
//Windows下利用Test And Set原子操作实现TAS操作
#else
# define TAS(l, n) os_atomic_test_and_set_uint((l), (n)) //非Windows下利用Test
And Set原子操作实现TAS操作（如Linux下调用__sync_lock_test_and_set()系统函数实现自旋锁，
```



```

PostgreSQL也同样地调用了类似的函数)
#endif /* _WIN32 */
#define CAS(l, o, n) os_val_compare_and_swap_uint((l), (o), (n))
//不同OS下利用Compare And Exchange原子操作实现CAS操作

```

InnoDB 主要定义了如下类型的系统锁：

```

typedef ib_mutex_t RsegMutex;           //保护回滚段的系统锁
typedef ib_mutex_t TrxMutex;            //保护事务块的系统锁，这是单个事务的事务块的内容保护
typedef ib_mutex_t UndoMutex;          //保护UNDO日志的系统锁
typedef ib_mutex_t PQMutex;            //保护PURGE QUERY的系统锁
typedef ib_mutex_t TrxSysMutex;         //保护整个InnoDB系统级的全局事务信息的系统锁，这是全局系统内所有在运行的事务的整体信息保护

typedef ib_mutex_t LockMutex;           //保护锁表的系统锁

```

“ib\_mutex\_t”又分别有多种可能，可能被前面描述的六种 Mutex 的其中之一根据宏定义进行选择而定义。

```

#ifdef HAVE_IB_LINUX_FUTEX
typedef PolicyMutex<TTASFutexMutex<DebugPolicy>> FutexMutex;
#endif /* HAVE_IB_LINUX_FUTEX */

typedef PolicyMutex<TTASMutex<DebugPolicy>> SpinMutex;
typedef PolicyMutex<TTASEventMutex<DebugPolicy>> SyncArrayMutex;

```

在 innodb.cmake 文件中定义如下内容：

```

IF(MUTEXTYPE MATCHES "event")
    ADD_DEFINITIONS(-DMutex_EVENT)
ELSEIF(MUTEXTYPE MATCHES "futex" AND DEFINED HAVE_IB_LINUX_FUTEX)
    ADD_DEFINITIONS(-DMutex_FUTEX)
ELSE()
    ADD_DEFINITIONS(-DMutex_SYS)
ENDIF()

typedef PolicyMutex<OSTrackMutex<DebugPolicy>> SysMutex;
typedef PolicyMutex<OSBasicMutex<DebugPolicy>> EventMutex;

#ifdef MUTEX_FUTEX
/** The default mutex type. */
typedef FutexMutex ib_mutex_t;
#define MUTEX_TYPE "Uses futexes"
#elif defined(Mutex_SYS)
typedef SysMutex ib_mutex_t;
#define MUTEX_TYPE "Uses system mutexes"
#elif defined(Mutex_EVENT)
typedef SyncArrayMutex ib_mutex_t;
#define MUTEX_TYPE "Uses event mutexes"

```

```
#else
#error "ib_mutex_t type is unknown"
#endif /* MUTEX_FUTEX */
```

## 2. InnoDB 定义的 Mutex 的作用

在许多对象（如数据缓冲区、字典表、系统锁表、双写缓冲区等）上、在许多操作作用的对象（如事务、回滚段等）上，InnoDB 都定义了很多系统锁，用以保护某个对象。这些系统锁，就是上一节讨论的 Mutex。定义好的 Mutex 的具体作用，详情参见表 11-5。

表 11-5 InnoDB 的系统级锁

类别	锁的名称	锁及作用
数据缓冲区	buf_pool	buf_pool->mutex, 保护 InnoDB 实例的 buffer pool
	buf_pool_zip	buf_pool->zip_mutex, 保护 InnoDB 实例的 buffer pool 中的压缩页
	buf_block_mutex	block->mutex, 保护 buffer 控制块
	flush_list	buf_pool->flush_list_mutex, 保护 InnoDB 实例的 buffer pool 中刷出 list
页面清理	page_cleaner	page_cleaner->mutex, 保护清理页面相关的对象 page_cleaner_t 和 page_cleaner_slot_t
双写缓冲区	buf_dblwr	buf_dblwr->mutex, 保护双写缓冲区
数据字典	dict_sys	dict_sys->mutex, 保护数据字典表, 并使得 CREATE TABLE 和 DROP TABLE 互斥
	dict_foreign_err	dict_foreign_err_mutex, 保护外键、唯一键错误信息所写的临时文件不被同时操作
GIS 使用的 R-tree	rtr_active_mutex	index->rtr_track->rtr_active_mutex, 保护处于 ACTIVE 状态的 “rtr_info” 列表
	rtr_ssn_mutex	index->rtr_ssn_mutex, ssn 是 Split Sequence Number 之意, 是 R-tree 索引上因页分裂时而赋予新页的序列号, 保护并发下多页分裂时 ssn 值的原子性
	rtr_match_mutex	rtr_info->matches->rtr_match_mutex, 保护页节点上匹配的记录
	rtr_path_mutex	rtr_info->rtr_path_mutex, 保护 R-tree 上的 “路径”, 路径是从根经过内部若干节点到达叶子的一条路
insert buffer	ibuf	ibuf_mutex, 保护插入缓存
	ibuf_bitmap	ibuf_bitmap_mutex, 保护插入缓存的 bitmaps
	ibuf_pessimistic_insert	ibuf_pessimistic_insert_mutex, 悲观地保护插入缓存的插入操作, 悲观的含义是严格限制并发操作即禁止并发的范围较大
事务相关	trx_sys	trx_sys->mutex, 保护全局事务的信息
	trx	trx->mutex, 保护单个事务的信息
	trx_undo	trx->undo_mutex, 保护单个事务的 UNDO 信息
	trx_pool	TrxPoolLock.m_mutex, 保护事务池
	trx_pool_manager	TrxPoolManagerLock.m_mutex, 保护事务池管理器



(续)

类别	锁的名称	锁及作用
事务锁	lock_sys	lock_sys->mutex, 保护事务锁的锁表
	lock_sys_wait	lock_sys->wait_mutex, 保护事务锁的锁表内部的部分成员
Spin Lock, InnoDB 称为读写锁	rw_lock_list	rw_lock_list_mutex, 保护读写锁对象列表即 rw_lock_list, 这个对象的锁是通过 rw_lock_create 宏所表示的 rw_lock_create_func() 传入的
	.rw_lock_debug	rw_lock_debug_mutex
	rw_lock_mutex	rw_lock_get_mutex(lock), 保护读写锁自身
数据统计	recalc_pool	recalc_pool_mutex, 保护 recalc_pool, recalc_pool 是存储后台进程需要做数据统计的表的 id 集合
REDO log buffer	log_sys	log_sys->mutex, 保护 REDO 日志的 buffer
	log_flush_order	log_sys->log_flush_order_mutex, 保护 REDO 日志 buffer 的刷出 List 有序刷出
回滚段	noredolog_rseg/redolog_rseg	rseg->mutex, 保护一个回滚段
记录索引修改的 buf	index_online_log	log->mutex, 保护修改索引的 buf
INFORMATION SCHEMA 的内部结构	row_drop_list	row_drop_list_mutex, 保护 DROP TABLE 的列表
	cache_last_read	cache->last_read_mutex
系统恢复	recv_sys	recv_sys->mutex, 保护恢复系统做 Apply 操作的 recv_sys_t 的内存结构
	recv_writer	recv_sys->writer_mutex, 在 recv_writer_thread 和 the recovery thread 之间协调刷出动作
队列	work_queue	wq->mutex, FIFO SPMC work queue, single producer, multiple consumers
反转索引, 即 FTS	fts_delete	cache->deleted_lock
	fts_optimize	cache->optimize_lock
	fts_doc_id	cache->doc_id_lock
	fts_bg_threads	bg_threads_mutex, 保护 bg_threads 和 fts_add_wq
	fts_pll_tokenize	psort_info[j].mutex, 并行排序
表空间	fil_system	fil_system->mutex, 保护表空间的 cache
PURGE 操作	purge_sys_pq	purge_sys->pq_mutex, 保护 PURGE 操作的控制块
自增对象	autoinc	table->autoinc_mutex, 保护表对象上的自增列
文件格式	file_format_max	file_format_max.mutex, 文件格式内存对象的保护
压缩页上的统计信息	page_zip_stat_per_index	page_zip_stat_per_index_mutex, 保护
页压缩	zip_pad_mutex	index->zip_pad.mutex, 保护 zip_pad_info_t 结构体, zip_pad_info_t 是在未被压缩的页面上面留有多少空间给压缩使用
系统监控	srv_innodb_monitor	srv_innodb_monitor_mutex, 保护系统监控信息, 如异步 IO 信息、日志刷出状态信息等



(续)

类别	锁的名称	锁及作用
DB Server	srv_sys	srv_sys->mutex, 保护 srv_sys_t 结构体, srv_sys_t 存放后台线程的相关信息, 表明线程的是否 ACTIVE 的状态等
	srv_sys_tasks	srv_sys->tasks_mutex, 保护 srv_sys_t 结构体的任务队列
	srv_monitor_file	srv_monitor_file_mutex, 保护临时的监控文件
	srv_dict_tmpfile	srv_dict_tmpfile_mutex, 保护临时的数据字典输出文件
	srv_misc_tmpfile	srv_misc_tmpfile_mutex, 保护临时的混合诊断输出文件
OS	thread_mutex	thread_mutex, 保护线程对象 (主要是 os_thread_count, 活动状态的线程计数器)
事件	event_mutex	os_event_mutex, 保护 InnoDB 的 condition
文件与 IO	os_file_seek_mutex	os_file_seek_mutexes[i], 保护对 OS 一层的文件的 seek 操作
	os_aio_mutex	array->mutex, 保护异步 IO 的操作, 操作有针对 Insert Buffer、REDO Log、Read array、Write array 等

### 3. InnoDB 定义的 Mutex 的用法

我们使用数据缓存区的 Mutex 来说明 Mutex 的用法。其他 Mutex 的用法都相似, 如下: 首先, Mutex 被创建并初始化的方法如下:

```
mutex_create("buf_pool", &buf_pool->mutex); //调用mutex_create()函数完成一个Mutex的创建和初始化。
```

其次, 使用两个配对使用的宏定义, 完成共享临界资源的加锁和解锁操作。

```
buf_pool_mutex_enter(buf_pool); //宏定义, 在buf_pool对象上的mutex上加锁
... //一些代码, 使得buf_pool的成员变量修改被保护起来
buf_pool_mutex_exit(buf_pool); //宏定义, 在buf_pool对象上的mutex上解锁。
注意要配对使用, 并注意前后的顺序
```

其中, buf\_pool\_mutex\_enter 等作为宏, 其定义类似如下内容:

```
#define buf_pool_mutex_enter(b) do { \
    ut_ad(!(b)->zip_mutex.is_owned()); \
    mutex_enter(&(b)->mutex); \
    //宏利用了mutex_enter(), 并把被保护的对象上的mutex加锁或解锁 \
} while (0)
```

再次, Mutex 的释放使用了 mutex\_free() 函数。

```
mutex_free(&buf_pool->mutex);
```

而不同类型的 Mutex 的子函数的实现方式, 存在差异, 以重要的 enter() 函数为例, 我们可以看出这些差异。如下分析几种不同的 Mutex。

#### (1) OSBasicMutex 的获取 Mutex 函数

```
void enter(
    ulint    max_spins, //最大的spin次数
    ulint    max_delay, //对于每一个spin最大的延迟
```





```

    const char* filename,
    uint line) UNIV_NOTHROW
{
    ut_ad(innodb_calling_exit || !m_freed);
#ifdef _WIN32 //可以看出，不管是Windows还是非Windows系统，参数max_spins和max_delay都没有被使用，即加锁不成功则被阻塞
EnterCriticalSection((LPCRITICAL_SECTION) &m_mutex);
#else
    int ret = pthread_mutex_lock(&m_mutex); //当pthread_mutex_lock()返回时，此互斥锁已被锁定。线程调用该函数让互斥锁上锁，如果该互斥锁已被另一个线程锁定和拥有，则调用该线程将阻塞，直到该互斥锁变为可用为止。这说明使用OSBasicMutex定义的Mutex，线程要么获得Mutex要么被阻塞。
    ut_a(ret == 0);
#endif /* _WIN32 */
}

```

## (2) OTrackMutex 的获取 Mutex 函数

```

void enter(
    uint max_spins, //最大的spin次数
    uint max_delay, //对于每一个spin最大的延迟
    const char* filename,
    uint line) UNIV_NOTHROW
{
#ifdef _WIN32
    bool locked = try_lock(); //第一次尝试加锁
    for (uint i = 0; !locked && i < max_spins; ++i) { //多次尝试加锁
        ut_delay(ut_rnd_interval(0, max_delay));
        //加锁不成功则有时间延迟，然后继续尝试加锁
        locked = try_lock();
        //继续尝试加锁
    }
    if (locked) {
        return;
    }
#endif /* _WIN32 */
    OSBasicMutex<Policy>::enter(
        //非Windows版本则调用OSBasicMutex<Policy>::enter，即加锁不成功则被阻塞
        max_spins, max_delay, filename, line);
    ut_ad(!m_locked);
    ut_d(m_locked = true);
}

```

## (3) TTASEventMutex 的获取 Mutex 函数

```

void enter(
    uint max_spins, //最大的spin次数
    uint max_delay, //对于每一个spin最大的延迟
    const char* filename,
    uint line) UNIV_NOTHROW

```



```

{
    if (!try_lock()) { //加锁不成功则等待
        spin_and_wait(max_spins, max_delay, filename, line);
    }
}

```

从上述三个类型的 enter() 函数的分析可以看出, 不同类型的 Mutex 的获取函数, 实现方式是有差异的。有的是加锁不成功则直接陷于阻塞如 OSBasicMutex, 有的是加锁不成功则尝试多次如 OTrackMutex, 这些方式, 都和 SpinLock 有差异, 可以仔细对比 PostgreSQL 的 SpinLock 相关内容。

### 11.2.3 其他锁

另外, MySQL Server 层的代码定义了一些锁操作, 如 thr\_rwlock.h 文件中, “native\_rw\_\*( )” 类的锁映射了操作系统的读写锁 (如 Windows 调用 AcquireSRWLockShared(), 非 Windows 调用 pthread\_rwlock\_rdlock()); “mysql\_rw\_\*( )” 类的锁是为 MySQL Server 层提供服务, 但间接调用了 “native\_rw\_\*( )” 类的锁, 如用于权限检查的 check\_grant\_db() 函数里就是这样的, 代码如下:

```

bool check_grant_db(THD *thd, const char *db) //检查SESSION上的用户在DB上的权限
{
    ...
    mysql_rwlock_rdlock(&LOCK_grant); //锁定LOCK_grant锁
    for (uint idx=0 ; idx < column_priv_hash.records ; idx++) //一些权限检查操作
    ...
    mysql_rwlock_unlock(&LOCK_grant); //解锁LOCK_grant锁, 与加锁配对使用
    ...
}

mysql_rwlock_t LOCK_grant //LOCK_grant锁的定义

```

其中, mysql\_table\_grant() 等函数也要使用同一个 LOCK\_grant, 阻塞了并发操作。这一类锁, 属于 MySQL Server 层, 和储存与事务引擎 InnoDB 没有关系。

## 11.3 事务锁之记录锁

InnoDB 除了支持用于系统级别的系统锁外, 还支持对会话 (用户会话、系统会话) 发起的事务在事务内部避免对用户数据的或表对象的元数据不一致而施加锁操作, 这样的锁操作遵循两阶段提交, 属于基于锁的并发控制封锁技术范畴。

如上所述, MySQL Server 和 InnoDB 事务锁主要分为两大类, 一是保护元数据的如元数据锁 (主要体现在 MySQL Server 层, 参见 11.4 节), 一是保护数据的如记录锁; 另外为了支持空间索引在 InnoDB 中引入谓词锁也数据保护数据的锁。





本节就事务锁中的记录锁展开讨论。

### 11.3.1 记录锁的基本数据结构

InnoDB 事务锁的基本数据结构包括锁的粒度、锁的种类、锁的基本结构体和锁的相容性等最基础的信息。如下逐一对这些基本内容进行介绍。

#### 1. 锁的粒度

InnoDB 的事务锁，支持四种基本的锁粒度（有的人称之为 mode），这四种锁的粒度比 Informix 锁的粒度粗（Informix 锁的粒度参见第 3 章）。

```
enum lock_mode {
    LOCK_IS = 0,    /* intention shared */    //意向共享锁
    LOCK_IX,        /* intention exclusive */    //意向排它锁
    LOCK_S,         /* shared */    //共享锁
    LOCK_X,         /* exclusive */    //排它锁
    LOCK_AUTO_INC,  /* locks the auto-inc counter of a table in an exclusive mode */
                                                           //自增锁。简称AI
    LOCK_NONE,      /* this is used elsewhere to note consistent read */
    LOCK_NUM = LOCK_NONE, /* number of lock modes */
    LOCK_NONE_UNSET = 255
};
```

说明：

- S 或 X 锁锁定的是行元组对象，但施加在页面的锁表中（参见 11.3.2 节）。
- IS 或 IX 锁锁定的是表对象，因为寻找元组时先要经过表对象，所以在表对象上施加意向锁。
- 当用户显式执行 LOCK TABLE 命令时，在表对象上显式加 S 或 X 锁。

#### 2. 锁的种类

InnoDB 的事务锁，支持六种基本的锁类型：

```
/* Precise modes */
#define LOCK_ORDINARY    0
//普通锁，与LOCK_GAP或LOCK_REC_NOT_GAP没有关联⊖，但是包括了间隙锁
#define LOCK_GAP        512    /*!< when this bit is set, it means that the //间隙锁
                                lock holds only on the gap before the record;
                                //记录之前的间隙被锁定，阻止记录被修改，也阻止记录前的间隙被插入
                                for instance, an x-lock on the gap does not
                                give permission to modify the record on which
                                the bit is set; locks of this type are created
                                when records are removed from the index chain
                                of records */
```

⊖ 没有关联的含义是他们之间不重叠使用不同时使用，但语义上都和间隙锁存在一定关系。



```

#define LOCK_REC_NOT_GAP 1024    /*!< this bit means that the lock is only on
//记录被锁定，记录之前的间隙不被锁定
        the index record and does NOT block inserts
//即阻止记录被修改，不阻止记录前的间隙插入
        to the gap before the index record;
//使用场景是当查询语句条件带有唯一键时施加的锁不带有间隙锁故不阻塞插入操作
        this is used in the case when we retrieve a record
        with a unique key, and is also used in
        locking plain SELECTs (not part of UPDATE
        or DELETE) when the user has set the READ
//只用于READ COMMITTED隔离级别，以模仿Oracle的读已提交级别的行为
        COMMITTED isolation level */
#define LOCK_INSERT_INTENTION 2048 /*!< this bit is set when we place a waiting
//插入意向锁，专门用于插入操作
        gap type record lock request in order to let
        an insert of an index record to wait until
        there are no conflicting locks by other
        transactions on the gap; note that this flag
        remains set when the waiting lock is granted,
        or if the lock is inherited to a neighboring
        record */
#define LOCK_PREDICATE      8192    /*!< Predicate lock */ //谓词锁，用于空间索引
#define LOCK_PRDT_PAGE      16384    /*!< Page lock */ //谓词锁相关的页锁

```

对于这些锁的类型的具体含义，在第5章5.2节有着详细说明，这里不再赘述。

但是，需要特别说明的是，这些种类，在逻辑上归属于谓词锁的范畴（但不要与11.6.2节的谓词锁在逻辑概念层面混淆，空间索引的谓词锁是本处提及的谓词锁的一部分），即为了实现谓词锁的语义以实现事务的隔离级别，才定义了这些锁的种类，因此这些种类没有和上节的锁的粒度放在一起去定义。这是需要在概念上一定要分清楚的事情。这一点，可以从如下函数定义中仔细体会。

```

sel_set_rec_lock( //Sets a lock on a record.此函数是在记录上设置锁的入口，对于所要施加的
//锁，用mode和type两个参数来标识（体会分开的用意和这些参数被赋值的情况）
    btr_pcur_t*   pcur,    /*!< in: cursor */
    const rec_t*   rec,    /*!< in: record */ //记录
    dict_index_t*  index,  /*!< in: index */
    const ulint*   offsets, /*!< in: rec_get_offsets(rec, index) */
    ulint          mode,    /*!< in: lock mode *///加锁的mode，即本书称为粒度，如X锁、S锁等
    ulint          type,    /*!< in: LOCK_ORDINARY, LOCK_GAP, or LOC_REC_NOT_GAP */
    //加锁的type，即本书称为种类，如LOCK_GAP等
    que_thr_t*     thr,    /*!< in: query thread */
    mtr_t*         mtr)    /*!< in: mtr */

```

对于 sel\_set\_rec\_lock() 的调用一个示例如下：

```

row_sel( //执行一个SELECT查询

```





```

sel_node_t*    node,    /*!< in: select node */
que_thr_t*     thr)     /*!< in: query thread */
{
    ...
    if (srv_locks_unsafe_for_binlog
        || trx->isolation_level<= TRX_ISO_READ_COMMITTED) {
//对于不同隔离级别, lock_type的值不同
        if (page_rec_is_supremum(next_rec)) {
            goto skip_lock;
        }
        lock_type = LOCK_REC_NOT_GAP;
    } else {
        lock_type = LOCK_ORDINARY;
    }

    err = sel_set_rec_lock(&plan->pcur,next_rec, index, offsets,
        node->row_lock_mode,lock_type, thr, &mtr);
    ...}

```

### 3. 锁的基本结构

InnoDB 事务锁的基本结构如下：

```

/** Lock struct; protected by lock_sys->mutex */
struct lock_t {
    trx_t*    trx;          /*!< transaction owning thelock */ //哪个事务用于此锁
    UT_LIST_NODE_T(lock_t) trx_locks; /*!< list of the locks of thetransaction */
    //事务已经申请到的事务的锁的双向列表
    dict_index_t*    index; /*!< index for a record lock */
    lock_t*          hash; /*!<hash chain node for a recordlock. The link node
in a singly linkedlist, used during hashing. */
    union {
        lock_table_t    tab_lock; /*!< table lock */ //表锁
        lock_rec_t      rec_lock; /*!< record lock */ //记录锁
    } un_member; /*!< lock details */ //union表明一个锁不能同时是表锁和记录锁,
    只能是其一
    ib_uint32_t    type_mode; /*!< lock type, mode, LOCK_GAP or
//锁的种类, 如上节的说明
        LOCK_REC_NOT_GAP,
        LOCK_INSERT_INTENTION,
        wait flag, ORed */
};

```

### 4. 锁的相容性

在数据引擎中，并发的交易之间存在锁竞争关系，所以加锁请求需要判断锁是否相容（如表 11-6 所示），如果不相容，则新申请的锁不能被授予，这样抑制了并发操作。



表 11-6 记录锁事务锁相容表

		Granted Mode, 已经授予的锁				
		AI	IS	S	IX	X
Requested Mode 正申请的锁	AI		Y		Y	
	IS		Y	Y	Y	
	S		Y	Y		
	IX		Y		Y	
	X					

说明:

□ “Y” 表示可以被授予锁, 空白则表示不可以授予新请求的锁。

但是, 同一个事务内部, 可能对同一个对象存在多次访问请求, 而多次访问请求可能意味着有不同类型的锁请求, 这时就不能通过锁的相容性列表进行判断, 而是需要使用锁的升级表(如表 11-7 所示)进行判断。

表 11-7 记录锁事务锁升级相容表

		Granted Mode, 已经授予的锁				
		AI	IS	S	IX	X
Requested Mode 正申请的锁	AI	Y				Y
	IS		Y	Y	Y	Y
	S			Y		Y
	IX				Y	Y
	X					Y

说明:

□ “Y” 表示可以被授予锁, 空白则表示不可以授予新请求的锁。

## 5. 显式锁与隐式锁

封锁机制是一种悲观的并发控制方法, 它并不是在发生冲突时才加锁或检测, 而是毫无商量地直接使用锁来互斥并发操作。但如果冲突的可能性很小, 有一些类型的锁是不必要施加的, 这样能够提高并发效率, 同时节约锁表等资源。

为此, InnoDB 从加锁的必要性上把锁分为两种, 一种称为显式锁, 一种称为隐式锁。

□ 显式锁 (Explicit Lock): 使用 “LOCK TABLES” 操作的过程中锁用到的锁, 都是显式锁。

○ 显式行锁, 加锁时会锁定(索引上的)记录和记录之前的间隙。可以参见 `set_explicit_duration_for_all_locks()` 函数。

○ 显式行锁, 加锁时也可能锁定(索引上的)记录但不包括记录之前的间隙。

□ 隐式锁 (Implicit Lock): 是 InnoDB 实现的一个延迟加锁的机制, 用来减少加锁的数量。这意味着隐式锁是一个逻辑概念上的锁, 其神存在其形不存在。

○ InnoDB 的隐式锁必定是排它锁。





- 对于隐式的排它锁，不能锁定间隙只能锁定（索引上的）记录<sup>⊖</sup>。这是因为隐式锁是针对被修改的记录（属于 Record 类型），所以不能有间隙（GAP）。没有为间隙位（gap bit）做过置位的显式锁可以施加在记录和间隙上。如果间隙位（gap bit）被置位，锁只施加在间隙上。
- INSERT 操作会在记录的系统列写入 `trx_id`，根据 `lock_clust_rec_some_has_impl()` 函数判断，这种情况下只含有隐式锁。
- 隐式锁可以向显式锁转换：
  - `lock_clust_rec_some_has_impl()`、`lock_sec_rec_some_has_impl()`、`lock_rec_has_expl()` 等函数帮助判断是否存在隐式锁或显式锁。
  - `lock_rec_convert_impl_to_expl()` 函数将隐式排它锁转换成显示排它锁。
  - InnoDB 的每条记录中都有一个隐含的系统字段 `trx_id`，这个字段存在于记录上，即聚簇索引（B+Tree）中。
  - 操作一条记录前，先要根据记录中的 `trx_id` 检查该事务是否是处于活动的状态（未提交或回滚）。
  - 如果是活动的事务且没有提交且不存在显式排它锁（`trx_rw_is_active()` 函数、事务的状态是否匹配 `TRX_STATE_COMMITTED_IN_MEMORY`、`lock_rec_has_expl()` 函数帮助判断），意味着有隐式锁需要转换为显式锁。
  - 调用 `lock_rec_add_to_queue()` 函数，实现加锁请求入加锁队列。
- 显式锁、隐式锁和索引：
  - 如果一个事务修改或插入一个索引记录，事务管理器会在索引记录（clustered index record）上加一个隐式的排它锁。此时，如存在二级索引（a secondary index）则在其记录上，事务管理器会加一个隐式的排它锁，二级索引的记录所在的页面上的事务 ID 值将大于等于正在加锁的事务的 ID，并且不能有显式的非间隙锁作用在二级索引的记录上。
- 其他情况：
  - 对于间隙锁，不同的事务不会同时作用在其上。插入操作不会产生间隙锁，查询操作如遇到间隙锁则需要等待，排它的间隙锁禁止插入操作插入到“间隙”里所以禁止了幻象异常现象。
  - 显式锁可以作用在记录上或一个页面的上边界（此时可以认为在一个记录上存在锁，并且其前有间隙锁，所以上边界是一个特殊情况默认在概念上带有间隙但是代码里不会明显指定加间隙锁）。当这样的记录被更新导致记录的大小（size）发生变化的时候，系统会临时显式锁定记录之后的下边界（通常情况下，下边界是不会被施加锁的）以做扩展使用。上边界和下边界参见表 11-8。

⊖ 加锁申请时的代码段为：`type_mode = (LOCK_REC | LOCK_X | LOCK_REC_NOT_GAP)`



表 11-8 索引页结构表

类型	成员
PAGE_HEADER 页头	PAGE_N_DIR_SLOTS
	PAGE_HEAP_TOP
	PAGE_N_HEAP
	PAGE_FREE
	PAGE_GARBAGE
	PAGE_LAST_INSERT
	PAGE_DIRECTION
	PAGE_N_DIRECTION
	PAGE_N_RECS
	PAGE_MAX_TRX_ID
	PAGE_HEADER_PRIV_END
页相关的其他信息	PAGE_LEVEL
	PAGE_INDEX_ID
	PAGE_BTR_SEG_LEAF
	PAGE_BTR_IBUF_FREE_LIST
	PAGE_BTR_IBUF_FREE_LIST_NODE
空闲	PAGE_BTR_SEG_TOP
	10
PAGE_DATA 页体 <sup>①</sup>	PAGE_OLD_INFIMUM / PAGE_NEW_INFIMUM, 最大下界
	record id1
	record id2
	record id3
	record ...
	record idN
	PAGE_OLD_SUPREMUM / PAGE_NEW_SUPREMUM, 最大上界, 此处加间隙锁阻止插入
	PAGE_OLD_SUPREMUM_END / PAGE_NEW_SUPREMUM_END

注：① “PAGE\_OLD\_INFIMUM / PAGE\_NEW\_INFIMUM” 到 “record id1” 逐渐到 “record idN” 再到 “PAGE\_OLD\_SUPREMUM / PAGE\_NEW\_SUPREMUM”，是一个从小到大的序列。

- 处于等待状态的记录锁可以是间隙锁，如果显式锁队列中靠前的位置不存在其他的冲突的锁请求则这个等待的锁请求就可以被授予。
- 在同一个记录上，不同事务不可能拥有相冲突的被授予的非间隙锁（non-gap locks），但是可以拥有冲突的间隙锁。
- 如果其他事务没有在一个记录上施加显式的锁，则插入记录（新数据对应新记录）到此记录之前的间隙里是被允许的。此时，不必关注那些锁的请求是被授予还是处于等待状态、间隙锁是被设置还是没有设置（设置的含义是 GAP 的 bit 标志位是否被置位）；但是，一个例外的情况是，当其他等待本事务（请求间隙锁的事务）的事务要执行插入操作将被忽略。另外，其他事务生成的隐含排它锁不会阻止插





入操作，这样能允许插入操作并发。

- 同一个事务内，如果记录上有排它锁，本事务发出的修改操作是被允许的。
- 如果某个事务施加过不带有间隙锁的显式锁、或者记录上存在一个隐含锁、或者其他事务没有在记录上存在有排它锁，读操作是被允许的。
- 对于二级索引，找出其上是否存在一个隐含的排它锁是很耗时的事情。数据库引擎不得不到相应的聚集索引的对应记录之前的版本上、查找是否有个处于活动状态的事务为二级索引的记录打过删除标识。
- 处于可重复读隔离级别的事务<sup>①</sup>已经读取过一些结果集（被读取过的记录会被施加共享锁，在页面的上边界<sup>②</sup>施加带有间隙锁的共享锁，可防止因插入操作而带来的幻象异常，即禁止了其他事务的修改操作抑制了并发），则本事务可以反复读取这个结果集，且能获得同样的结果（本事务自己没有修改过结果集中的数据）。这种情况下，索引页面的分裂、合并或者因记录被废弃而页面被去除并不影响上述操作，这是因为如果一个记录或页面的上边界被移动，下一个记录会继承间隙锁，因此同样能够禁止插入记录（数据）到同一个间隙上。
- 如果按照字母序排序（因为遍历的是索引，而索引是单向有序的），其中最大的元组被去掉（删除或修改），则会发生锁等待（本事务处于等待状态），这是因为下一个记录会继承间隙锁。

### 11.3.2 记录锁

记录锁是事务锁的主要类型的锁，是 InnoDB 实现并发控制技术的核心。本节着重讲述记录锁涉及的内容，与并发控制理论相关的内容，请阅读第 2 章相关部分。

#### 1. 记录锁的数据结构

记录锁的结构体很简单，描述了一个页面上哪些记录被加锁，注意是页面而不是元组。

```
/** Record lock for a page */
struct lock_rec_t { 记录锁
    ib_uint32_t    space; /*!< space id */           //记录在哪个表空间中
    ib_uint32_t    page_no; /*!< page number */       //记录在哪个页面中
    ib_uint32_t    n_bits; /*!< number of bits in the lockbitmap; NOTE: the lock
    bitmap is placed immediately after the lock struct */
}; //本结构体之后紧跟着一个位图，标记了一个页面中有哪些记录被加锁。这说明记录锁不是施加在记录
上的，而是施加在页面上的，但不是页锁
```

#### 2. 记录锁的生命周期

记录锁的生成和锁结构体内容的赋值等工作，由专门的函数负责完成。

- 
- ① 这种情况的事务操作被称为“a read cursor”，即可以反复从这个cursor上读取相同的数据，不多也不少，就是同一个结果集，能做到这点的，是可重复读这个隔离级别；而序列化虽然也能做到反复读取到同一个结果集但是只有读读并发无其他并发所以导致并发度太低。
  - ② 上边界，supremum，是一个页面上元组的最后段的部分，是最大的上界。参见表11-8。

如下代码段是记录锁初始化的过程。

```
lock_t*
RecLock::lock_alloc( //生成一个记录锁
    trx_t*      trx,      //为指定的事务生成一个记录锁，由此可见记录锁绑定在事务上
    dict_index_t* index,
    ulint       mode,      //锁的模式
    const RecID& rec_id, //在哪个记录上生成一个记录锁。事务、记录锁、记录三者建立起对应关系
    ulint       size)
{...
    lock_t*      lock;
    ...
    lock->trx = trx;
    lock->index = index;
    /* Setup the lock attributes */
    lock->type_mode = LOCK_REC | (mode & ~LOCK_TYPE_MASK); //标明是记录锁
    ...
    rec_lock.space = rec_id.m_space_id;
    rec_lock.page_no = rec_id.m_page_no; //至此，事务、记录锁、记录三者建立起对应关系
    /* Set the bit corresponding to rec */
    lock_rec_set_nth_bit(lock, rec_id.m_heap_no); //在记录锁上设置本记录对应的锁标志位
    ...
}
```

在记录锁被创建的时候，意味着 DML 等语句操作记录的时候，需要在被操作的记录上施加记录锁以保护记录不被并发的其他事务破坏数据的一致性。所以如图 11-2 所示，可以看出 lock\_alloc() 函数多是在 INSERT、UPDATE、DELETE 等显式的数据被插入、修改、删除的情况下，才施加记录锁。另外一些记录锁被施加的时候，是发生了诸如页面分裂导致记录被移动，因而需要锁定记录，避免数据不一致。

记录锁被创建之后，又被 lock\_add() 存入事务包括的记录锁的 Hash 锁表上。这样，在全局锁表内，就保存有了多个记录锁。

```
//Add the lock to the record lock hash and the transaction's lock list
void
RecLock::lock_add(lock_t* lock, bool add_to_hash)
{...
    if (add_to_hash) {
        ulint key = m_rec_id.fold();
        ++lock->index->table->n_rec_locks;
        HASH_INSERT(lock_t, hash, lock_hash_get(m_mode), key, lock);
        //把lock锁存入lock_hash_get(m_mode)指定的Hash锁表内
        //Hash锁表内共有三种，分别是记录锁Hash表、谓词锁Hash表、页锁Hash表，在lock_sys_t中定义，
        参见11.1.2节
    }
    ...
    UT_LIST_ADD_LAST(lock->trx->lock.trx_locks, lock); //把锁lock注册到事务的锁列表内，对应lock_rec_dequeue_from_page()中的UT_LIST_REMOVE
}
```



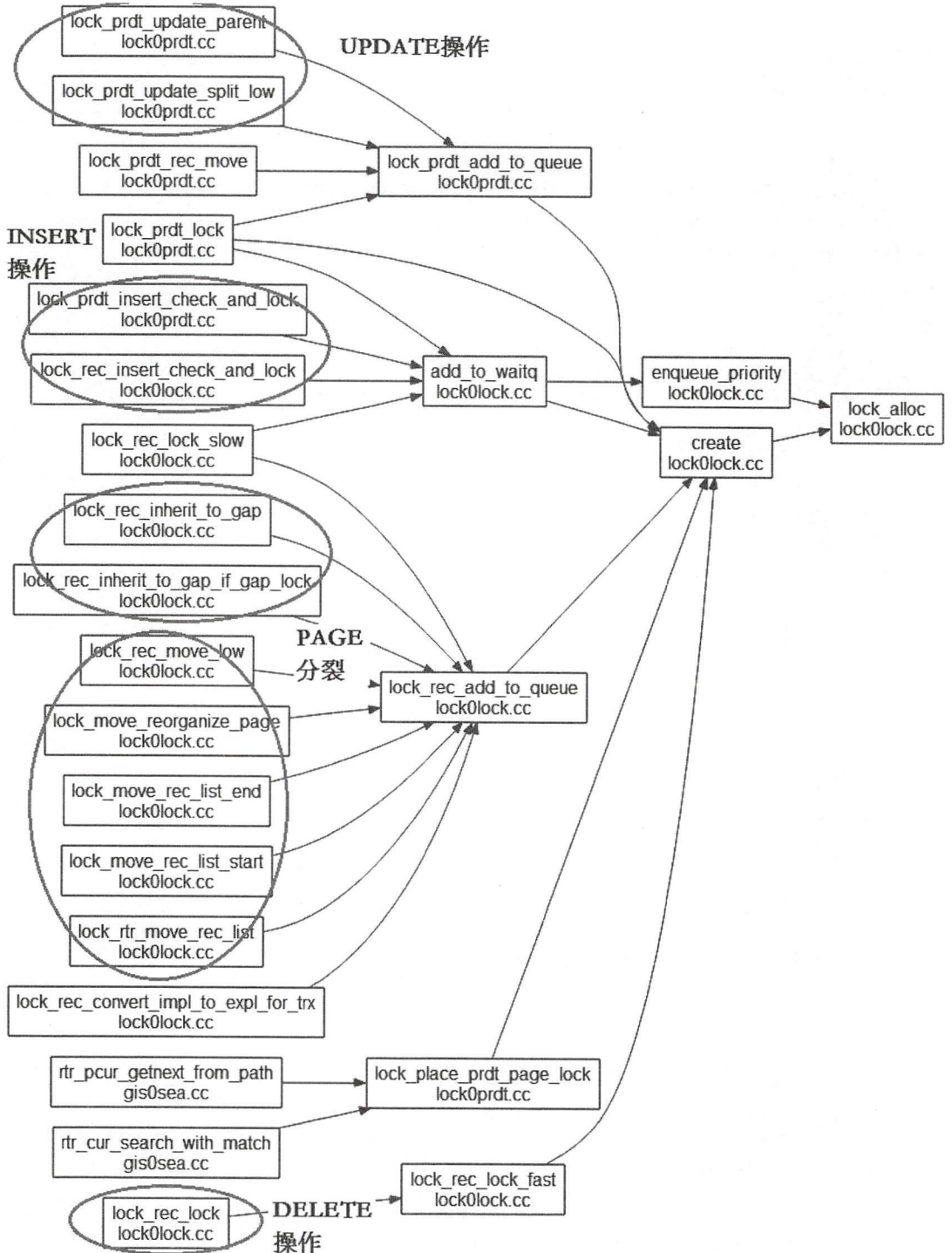


图 11-2 生成记录锁的上下文图

记录锁被使用完毕后，随着事务的结束，被 `lock_release()->lock_rec_dequeue_from_page()` 释放。

```
lock_rec_dequeue_from_page( //1 去掉记录锁。 2 为等待锁的事务施加锁
    lock_t*      in_lock) /*!< in: record lock object: all record locks which
    are contained in this lock object are removed;
    transactions waiting behind will get their lock requests
    granted, if they are now qualified to it */
{...
    in_lock->index->table->n_rec_locks--;

    lock_hash = lock_hash_get(in_lock->type_mode); //获得记录锁对应的Hash表
    HASH_DELETE(lock_t, hash, lock_hash,          //从Hash表中去掉记录锁
        lock_rec_fold(space, page_no), in_lock);

    UT_LIST_REMOVE(trx_lock->trx_locks, in_lock);
    //同时从事务锁表中去掉记录锁。对应RecLock::lock_add()中的UT_LIST_ADD_LAST
    ...
}
```

当申请锁发生错误时，逐层返回的错误信息被 `row_mysql_handle_errors()` 函数处理，相关内容请参见 11.3.2 节。

### 3. 记录锁的施加

InnoDB 为上层带有语义的加锁操作提供四个函数，进行语义级别的加锁：

```
lock_clust_rec_read_check_and_lock()    //因为读操作在主键索引记录上加记录锁
lock_clust_rec_modify_check_and_lock()  //因为修改（更新、删除）操作在主键索引记录上加记录锁
lock_sec_rec_modify_check_and_lock()    //因为读操作在二级索引记录（secondary index
record）上加记录锁
lock_sec_rec_modify_check_and_lock()    //因为修改（删除）操作在主键索引记录（secondary
index record）上加记录锁
```

这四个函数直接调用了底层的没有语义的记录锁的加锁函数 `lock_rec_lock()`，接受上层参数指定的加锁要求，然后进行加锁。加锁的机制有两种，在 InnoDB 中称为快速加锁和慢速加锁两种方式。

```
/*
*****
Tries to lock the specified record in the mode requested. If not immediately possible,
enqueues a waiting lock request.
This is a low-level function which does NOT look at implicit locks! Checks lock
compatibility within explicit locks.
This function sets a normal next-key lock, or in the case of a page supremum
record, a gap type lock.
@return DB_SUCCESS, DB_SUCCESS_LOCKED_REC, DB_LOCK_WAIT, DB_DEADLOCK, or DB_QUEUE_
THR_SUSPENDED */
```



```

static
dberr_t
lock_rec_lock(    //施加记录锁
    bool          impl, /*!< in: if true, no lock is set if no wait is necessary:
                        we assume that the caller will set an implicit lock */
    uint          mode, /*!< in: lock mode: LOCK_X or LOCK_S possibly ORed to
                        either LOCK_GAP or LOCK_REC_NOT_GAP */
    const buf_block_t* block, /*!< in: buffer block containing the record */
    uint          heap_no, /*!< in: heap number of record */
    dict_index_t* index, /*!< in: index of record */
    que_thr_t*    thr) /*!< in: query thread */
{
    ...
    /* We try a simplified and faster subroutine for the most common cases */
    switch (lock_rec_lock_fast(impl, mode, block, heap_no, index, thr)) {
        //尝试快速加锁, 如果失败, 才调用更严格的lock_rec_lock_slow()
        case LOCK_REC_SUCCESS:
            return(DB_SUCCESS); //快速加锁成功, 锁存在
        case LOCK_REC_SUCCESS_CREATED:
            return(DB_SUCCESS_LOCKED_REC); //快速加锁成功, 锁是新创建的
        case LOCK_REC_FAIL:
            return(lock_rec_lock_slow(impl, mode, block, heap_no, index, thr));
            //需要做更多判断才能确定的加锁机制, 因多做工作所以“slow”慢
    }
    ...
}

```

快速加锁, 是查看请求的记录所对应的缓存页面是否有相应的锁信息, 如没有则直接创建一个记录锁, 如果有则确认此锁是否可用。这样的判断方式谓之“fast”。而要想理解“fast”则应对比“slow”的方式。

```

/*****
This is a fast routine for locking a record in the most common cases:
there are no explicit locks on the page, or there is just one lock, owned by this
transaction, and of the right type_mode.
This is a low-level function which does NOT look at implicit locks! Checks lock
compatibility within explicit locks.
This function sets a normal next-key lock, or in the case of a page supremum
record, a gap type lock.
@return whether the locking succeeded */
UNIV_INLINE
lock_rec_req_status
lock_rec_lock_fast(...) //参数省略, 同上层的lock_rec_lock()函数
{
    ...
    lock_t*    lock = lock_rec_get_first_on_page(lock_sys->rec_hash, block);
    //从全局记录锁表中获得数据块对应的锁的信息
    trx_t*     trx = thr_get_trx(thr);

```

```

lock_rec_req_status    status = LOCK_REC_SUCCESS;

if (lock == NULL) {    //全局记录锁表中不存在数据块对应的锁的信息，则新建一个记录锁
    if (!impl) {
        RecLock rec_lock(index, block, heap_no, mode);
        //记录锁，定义在指定的索引index上
        /* Note that we don't own the trx mutex. */
        rec_lock.create(trx, false); //新建一个记录锁
    }
    status = LOCK_REC_SUCCESS_CREATED;
} else { //全局记录锁表中存在数据块对应的锁的信息
    trx_mutex_enter(trx);
    if (lock_rec.get_next_on_page(lock)
        //存在其他的记录锁(注意本函数是get_next，不是get_first)，则锁不唯一，可能存在竞争
        || lock->trx != trx //不是同一个事务
        || lock->type_mode != (mode | LOCK_REC) //不是记录锁
        || lock_rec.get_n_bits(lock) <= heap_no) {
        status = LOCK_REC_FAIL;
        //这种情况下，需要进入到lock_rec_lock_slow()的慢函数中进行更为详细地判断
    } else if (!impl) { //锁存在且不是隐含的锁，是显式加锁，则设置对应的标志位为加锁状态
        /* If the nth bit of the record lock is already set then we do not set
           a new lock bit, otherwise we do set */
        if (!lock_rec.get_nth_bit(lock, heap_no)) {
            lock_rec.set_nth_bit(lock, heap_no); //设置对应的标志位为加锁状态
            status = LOCK_REC_SUCCESS_CREATED;
        }
    }
    trx_mutex_exit(trx);
}
return(status);
}

```

“slow”的实现方式如下：

```

/*****
This is the general, and slower, routine for locking a record. This is a low-level
function which does NOT look at implicit locks!
Checks lock compatibility within explicit locks.
This function sets a normal next-key lock, or in the case of a page supremum
record, a gap type lock.
@return DB_SUCCESS, DB_SUCCESS_LOCKED_REC, DB_LOCK_WAIT, DB_DEADLOCK,
or DB_QUE_THR_SUSPENDED */
static
dberr_t
lock_rec_lock_slow(...) //参数省略，同上层的lock_rec_lock()函数
{...
    trx_mutex_enter(trx);

```



```

if (lock_rec_has_expl(mode, block, heap_no, trx)) {
    //是否在同一事务内存在更强的锁
    /* The trx already has a strong enough lock on rec: donothing */
    err = DB_SUCCESS; //在同一事务内存在更强的锁，则不需要申请新锁，这是锁的升级相容
                        问题，参见表11-7
} else {
    const lock_t* wait_for = lock_rec_other_has_conflicting(mode, block, heap_no, trx);
    //本事务trx是否被其他有显式锁的事务阻塞
    if (wait_for != NULL) { //存在有冲突的锁，只能等待（新建锁，但新建的锁需要进入
                            等待队列）

        /* If another transaction has a non-gap conflicting request in the
        queue, as this transaction does not
        have a lock strong enough already granted on the record, we may have to wait. */
        RecLock rec_lock(thr, index, block, heap_no, mode);
        err = rec_lock.add_to_waitq(wait_for); //新锁进入等待队列。并进行死锁判
        断，死锁判断将是一个耗时的过程，“slow”在此体现⊖
    } else if (!impl) {
        /* Set the requested lock on the record, note that we already own the
        transaction mutex. */
        lock_rec_add_to_queue(LOCK_REC | mode, block, heap_no, index,
        trx, true); //不是隐含锁，则把加锁请求加入等待队列
        err = DB_SUCCESS_LOCKED_REC;
    } else {
        err = DB_SUCCESS;
    }
}
}
trx_mutex_exit(trx);
return(err);
}

```

从上面的分析我们可以看出，所谓的加锁的“快”和“慢”，合起来就是一个完整的加锁过程，包括了加锁过程的所有情况，加锁过程中经常发生的情况被放到了“快”的函数中，而“慢”的函数因为要做死锁检测，可能会相对较慢。

#### 4. 记录锁的冲突判断

上一节，讨论了施加新锁因等待而需要进入等待队列，但是没有讨论什么情况下才会发生这样的事件，本节来看看锁冲突的条件。

```

/**Checks if some other transaction has a conflicting explicit lock request in the
queue, so that we have to wait.
@return lock or NULL */
static
const lock_t*

```

⊖ 互联网的一些企业，通过禁止掉死锁检测来提升性能（尤其是在热点更新场景），性能提升非常明显，极端高并发下，甚至能带来数倍的提升。

```

lock_rec_other_has_conflicting(
    ulint      mode,      /*!< in: LOCK_S or LOCK_X, possibly ORed to LOCK_GAP
                           or LOC_REC_NOT_GAP, LOCK_INSERT_INTENTION */
    const buf_block_t* block, /*!< in: buffer block containing the record */
    ulint      heap_no, /*!< in: heap number of the record */
    const trx_t*   trx) /*!< in: our transaction */
{
    const lock_t*   lock;
    ut_ad(lock_mutex_own());
    bool is_supremum = (heap_no == PAGE_HEAP_NO_SUPREMUM);
    //遍历全局锁表lock_sys中的记录锁表rec_hash, 查阅准备施加的锁(参数mode、block等表明新
    //锁的情况)是否被允许
    for (lock = lock_rec_get_first(lock_sys->rec_hash, block, heap_no);
         lock != NULL;
         lock = lock_rec_get_next_const(heap_no, lock)) { //lock是已经持有的锁
        if (lock_rec_has_to_wait(trx, mode, lock, is_supremum)) {
            //同一个事务则不存在冲突, 返回值为NULL; 否则, 返回持锁者
            return(lock);
        }
    }
    return(NULL);
}

```

在 lock\_rec\_has\_to\_wait() 函数中, 使用到了 lock\_mode\_compatible() 函数, 该函数调用 lock\_compatibility\_matrix 数组对锁的相容性(兼容性)做出判断, 即利用与表 11-6 所示相似的内容对申请施加的锁与已经被授予的锁之间的相容性做出判断。具体判断方式如下:

```

/*****
Checks if a lock request for a new lock has to wait for request lock2.
@return TRUE if new lock has to wait for lock2 to be removed */
UNIV_INLINE
ibool
lock_rec_has_to_wait(
    const trx_t*   trx,      /*!< in: trx of new lock */ //准备申请锁的事务
    ulint          type_mode, /*!< in: precise mode of the new lock to set: LOCK_S or LOCK_X,
                               possibly ORed to LOCK_GAP or LOCK_REC_NOT_
LOCK_INSERT_INTENTION */
    const lock_t*   lock2,    /*!< in: another record lock; NOTE that it is
                               assumed that this has a lock bit //已经分配的锁
                               set on the same record as in the new lock
                               we are setting */
    bool            lock_is_on_supremum) /*!< in: TRUE if we are setting the lock on
the 'supremum' record of an index page:
                               we know then that the lock request is
really for a 'gap' type lock */
{...

```



```

if (trx != lock2->trx //不是同一个事务。如果是同一个事务，则返回FALSE表明不存在冲突
    && !lock_mode_compatible(static_cast<lock_mode>(
        //不相容，即存在冲突；不同事务但锁相容，则返回FALSE表明不存在冲突
        LOCK_MODE_MASK & type_mode), lock_get_mode(lock2))) {
    /* We have somewhat complex rules when gap type record locks cause waits */
    //如下四条规则，放宽了限制，使得冲突减少。注意，这是提高并发度的方法
    if ((lock_is_on_supremum || (type_mode & LOCK_GAP))
        //规则1: 申请锁者申请GAP锁，不是插入操作，则锁不冲突不需要等待
        && !(type_mode & LOCK_INSERT_INTENTION)) {
        /* Gap type locks without LOCK_INSERT_INTENTION flag do not need to
           wait for anything.
           This is because different users can have conflicting lock types on gaps. */
        //这说明GAP上允许并发加锁
        return (FALSE);
    }

    if (!(type_mode & LOCK_INSERT_INTENTION) //规则2: 不是插入操作，且持锁者带有GAP
        //锁，则锁不冲突不需要等待
        && lock_rec_get_gap(lock2)) {
        /* Record lock (LOCK_ORDINARY or LOCK_REC_NOT_GAP) does not need to
           wait for a gap type lock */
        return (FALSE);
    }

    if ((type_mode & LOCK_GAP) //规则3: 申请锁者申请GAP锁，而持锁者不持有GAP锁，则锁
        //不冲突不需要等待
        && lock_rec_get_rec_not_gap(lock2)) {
        /* Lock on gap does not need to wait for a LOCK_REC_NOT_GAP type lock */
        return (FALSE);
    }

    if (lock_rec_get_insert_intention(lock2)) { //规则4: 持锁者持有插入意向锁，不阻塞任何锁
        return (FALSE);
    }
    return (TRUE);
}
return (FALSE);
}

```

从 `lock_rec_has_to_wait()` 函数的分析可以看出，决定记录锁的相容性不仅是相容性表（表 11-6），还有上述四条规则，他们共同决定了新申请的锁和已经授予的锁之间的相容关系。而 GAP 锁上通常是允许并发操作的，只是禁止了 INSERT 插入操作，这其实是在禁止幻象异常，所以 GAP 锁的主要作用是用于防止幻象异常。

下面一节我们来看迟滞加锁的情况，迟滞加锁，意味着可能有冲突，需要先过滤掉一些情况，然后再明确进行死锁检测。

## 5. 记录锁的等待与受害者判断

迟滞加锁，是因为锁不能立刻获得，即有锁存在导致新的加锁申请只能等待，或者有些情况下，死锁发生不得不进行死锁检测以挑选出一个受害者。受害者被挑选是有规则的，本节后半部分会就受害者的选取进行讨论。

锁等待的相关情况，主要是 `add_to_waitq()` 和 `enqueue_priority()` 函数。`add_to_waitq()` 统领着锁等待和受害者挑选的代码与逻辑。

先看 `add_to_waitq()` 函数的代码和分析：

```
/**
 * Enqueue a lock wait for normal transaction. If it is a high priority
 * transaction then jump the record lock wait queue
 * and if the transaction at the head of the queue is itself waiting roll it back,
 * also do a deadlock check and resolve.
 * @param[in, out] wait_for The lock that the joining transaction is waiting for
 * // 本锁被其他事务阻塞，所以本锁正等待其他事务完成
 * @param[in] prdt Predicate [optional] // 如果参数空缺，则参数值为 NULL
 * @return DB_LOCK_WAIT, DB_DEADLOCK, or DB_QUE_THR_SUSPENDED, or DB_SUCCESS_LOCKED_REC;
 * DB_SUCCESS_LOCKED_REC means that there was a deadlock, but another transaction was chosen
 * as a victim, and we got the lock immediately: no need to wait then */
dberr_t
RecLock::add_to_waitq(const lock_t* wait_for, const lock_prdt_t* prdt①)
// 使得本事务处于等待状态的锁是 wait_for
{
    ...
    m_mode |= LOCK_WAIT; // 因为本事务正等待其他事务，所以加标志 LOCK_WAIT
    /* Do the preliminary checks, and set query thread state */
    prepare();
    ...
    if (wait_for->trx->mysql_thd == NULL) {
        // mysql_thd 值为 NULL 表明是 InnoDB 引擎内部发起的事务，持有锁的内部事务不能作为受害者
        victim_trx = NULL;
    } else {
        /* Currently, if both are high priority transactions then the requesting
         * transaction will be rolled back. */
        victim_trx = trx_arbitrate(m_trx, wait_for->trx); // 事务优先级相同时，发起加锁
        // 请求的事务为受害者；否则，事务优先级低者为受害者
    }

    if (victim_trx == m_trx || victim_trx == NULL) {
        // 受害者是本事务，或者没有受害者（InnoDB 引擎内部发起的事务）
        /* Ensure that the wait flag is not set. */
        lock = create(m_trx, true, prdt); // 明知本事务可能被回滚，也需要创建出这个事务对应的锁
    }
}
```

① `lockpriv.h` 文件中的函数定义为：`dberr_t add_to_waitq(const lock_t* wait_for, const lock_prdt_t* prdt = NULL)`，`prdt` 可以缺省，所以调用 `add_to_waitq()` 的几处代码多是使用一个参数。被调用时如果只有一个参数，则 `prdt` 的值为 `NULL`。



```

/* If a high priority transaction⊖ has been selected as a victim there
   is nothing we can do. */
//本事务是高优先级事务，且不是InnoDB引擎内部发起的事务，被选为受害者，本事务只能回滚
if (trx_is_high_priority(m_trx) && victim_trx != NULL) {
//本事务因被回滚需要去掉其锁（注意条件“victim_trx == m_trx”）
    lock_reset_lock_and_trx_wait(lock); //因要把本事务回滚，所以先把本事务锁上的
    等待相关的标志去掉
    lock_rec_reset_nth_bit(lock, m_rec_id.m_heap_no); //同上
...
    return(DB_DEADLOCK); //有死锁发生啦！（已经能明确知道有死锁发生，不用进一步
                          进行死锁检测）⊕
}
} else if ((lock = enqueue_priority(wait_for, prdt)) == NULL) {
//当前事务不是受害者，lock为NULL意味着不用新加锁，锁已经被授予了
/* Lock was granted */
return(DB_SUCCESS); //加锁成功
}
ut_ad(lock_get_wait(lock)); //至此，已经明确知道必有锁等待，只好使用等待图进行死锁检测了
dberr_t err = deadlock_check(lock); //进行死锁检测，参见11.3.2.6节
ut_ad(trx_mutex_own(m_trx));
return(err);
}

```

再看，成为受害者的情况，从上述代码分析中能够看出：

- ❑ 首先，持有锁的 InnoDB 内部发起的事务不会被选为受害者。
- ❑ 其次，`trx_arbitrate()`<sup>⊕</sup> 函数表明：
  - ❑ 申请锁者是内部事务，持锁者优先级高则申请锁的事务为受害者；否则不挑选受害者。
  - ❑ 持锁者是内部事务，申请锁者优先级高则持锁的事务为受害者；否则不挑选受害者。
  - ❑ 如都不是内部事务，事务优先级相同时，发起加锁请求的事务为受害者；否则，优先级低者为受害者。
- ❑ 最后，当前事务是高优先级事务，且不是 InnoDB 引擎内部发起的事务，则表明有死

⊖ 高优先级事务的设置方式：

```

START TRANSACTION HIGH_PRIORITY;
UPDATE t1 SET c1=2 WHERE c1=0;
COMMIT;

```

高优先级事务的一些信息参见：<http://dev.mysql.com/worklog/task/?id=6835>

其他一些设置优先级的方式如下：

HIGH\_PRIORITY在SELECT和INSERT操作中使用，如：

```
SELECT HIGH_PRIORITY * FROM t1;
```

LOW\_PRIORITY在INSERT和UPDATE操作中使用，如：

```
UPDATE LOW_PRIORITY t1 SET field1=100 WHERE field1=1;
```

⊕ 这种情况下，上层函数会做符合各自逻辑的处理。如上层的一种调用是调用`dict_stats_rename_table()`做表的重命名操作，遇到死锁存在，上层的逻辑就是不断循环等待，直至尝试成功。

⊕ 通过`check_and_resolve()`做死锁检测时，也需要调用本函数，所以通过这个函数找出受害者的规则是一样的。

锁发生。如何处理，则交给上层的操作进行。

第三看 `enqueue_priority()` 函数。受害者不是当前事务，则需要调用 `enqueue_priority()` 函数围绕持锁者进行处理，处理又分为两种情况：第一种是一个特殊情况，持锁者也处于等待状态，引发持锁者和申请锁者等待的第三个事务将被回滚；第二种是第一种特殊情况之外的其他情况，锁将被加入锁相关队列，到死锁检测阶段进行识别判断。源码分析如下：

```
/**
 * Enqueue a lock wait for a high priority transaction, jump the record lock
 * wait queue and if the transaction at the head of the queue is itself waiting roll it back.
 * @param[in, out] wait_for The lock that the the joining transaction is waiting for
 * @return NULL if the lock was granted */
lock_t*
RecLock::enqueue_priority(const lock_t* wait_for, const lock_prdt_t* prdt)
//wait_for是持锁者
{
    /* Create the explicit lock instance and initialise it. */
    //下面的代码要使用到这里创建的lock
    lock_t*lock = lock_alloc(m_trx, m_index, m_mode, m_rec_id, m_size);
    //lock的属主是当前事务，在本函数尾部，可能被加入记录锁hash表中
    ...
    bool read_only = wait_for->trx->read_only;
    bool waiting = wait_for->trx->lock.que_state == TRX_QUE_LOCK_WAIT;
    //持锁者也在等待其他事务
    //如下注释表明两种情况。但一共有三种情况。
    //情况1： 阻塞当前事务的持锁者也在等待其他的锁，那么阻塞持锁者的事务要被回滚（有了级联阻塞
    的味道，T1等T2，T2等T3，则回滚T3）。
    // 情况1的例外情况： 除非持锁者和当前事务等待的是同一个锁。此种情况下，持锁者会被kill掉，
    即回滚。
    /* If the transaction that is blocking m_trx is itself waiting then we kill it
    in this method,
        unless it is waiting for the same lock that m_trx wants. For the latter
        case we kill it before doing the lock wait.
    //情况2： 阻塞当前事务的持锁者没有在等待其他的锁，但是个只读得事务，持锁者不被回滚。
    If the transaction is not waiting but is a read-only transaction started with
    START TRANSACTION READ ONLY then we wait for it. */
    bool kill_trx; //如下的if...else...语句为情况1和情况2进行条件判断
    if (waiting) { //情况1
        ut_ad(wait_for->trx->lock.wait_lock != NULL);
        /* Check if "wait_for" trx is waiting for the same lock and we can roll it
        back asynchronously. */
        kill_trx = wait_for->trx->lock.wait_lock != wait_for
        //确定是属于：“情况1”还是“情况1的例外情况”
        && !read_only
        && !(wait_for->trx->in_
innodb & TRX_FORCE_ROLLBACK_DISABLE);
```



```

} else if (read_only) { //情况2
    /* Wait for running read-only transactions */
    kill_trx = false;
} else { //情况3: 预先设定了禁止事务回滚的标识
    /* Rollback any running non-ro blocking transactions */
    kill_trx = !(wait_for->trx->in_innodb&TRX_FORCE_ROLLBACK_DISABLE);
}

/* Move the lock being requested to the head of the wait queue so that if the transaction that
   we are waiting for is rolled back we get dibson the row. */
//把参数lock指定的锁放入等待队列首位(head指向的第一个lock)和事务的锁list中, 打上回滚标志
jump_queue(lock, wait_for, kill_trx); //kill_trx值为TRUE, 才会为持锁者wait_for打
上回滚标志。注意这样才能满足下一个if语句的条件
if (waiting && kill_trx) {... //必是情况1但不是情况1的例外情况
    set_wait_state(lock);
    lock_set_lock_and_trx_wait(lock, m_trx); //让当前事务的锁处于等待状态
    /* Rollback the transaction that is blocking us. It should be the one that is
       at the head of the queue.
       Note this doesn't guarantee that our lock will be granted.
       We will kill other blocking transactions later in trx_kill_blocking(). */
    rollback_blocking_trx(wait_for->trx->lock.wait_lock⊖);
    //回滚导致持锁的事务等待的事务
    ...
    /* There is no guaranteed that the lock will have been granted even if we
       were the first in the queue. There could be other
       transactions that hold e.g., a granted S lock but are waiting for
       another lock. They will be rolled back later. */
    return(lock_get_wait(lock) ? lock : NULL); //返回还处于等待状态的当前事务的
    锁; 如锁被授予, 则返回NULL
} else {... //情况1之外的其他情况(持锁者没有被阻塞), 暂不需要回滚事务, 把锁加入记录锁
hash表和事务的锁表中
    lock_add(lock, false); //被加到记录锁hash表中的新锁, 将在下阶段的死锁检测中被判断
}

/* This state should not change even if we release the wait_for->trx->mutex.
   These can only change if we release the lock_sys_t::mutex. */
ut_ad(version == wait_for->trx->version);
ut_ad(read_only == wait_for->trx->read_only);
return(lock);
}

```

## 6. 记录锁的死锁检测

记录锁之间, 可能造成死锁, 所以需要在记录锁之间进行死锁检测。而死锁的成因,

⊖ 注意体会InnoDB对wait\_lock的注释: if trx execution state is TRX\_QUE\_LOCK\_WAIT, this points to the lock request, otherwise this is NULL

在 2.2.1 节进行过详细讨论，死锁的理论基础请参考 2.2.1 节。对于 InnoDB 而言，死锁可归于 InnoDB 使用的并发控制技术是基于锁的并发控制技术<sup>①</sup>，且又满足了造成死锁的四个条件。

InnoDB 使用 `deadlock_check()` 对记录锁进行死锁检测，步骤较为简单，第一步是死锁处理，即找出死锁并回滚受害者事务；第二步是检查死锁处理的结果。

```
/**
Check and resolve any deadlocks
@param[in, out] lock          The lock being acquired
@return DB_LOCK_WAIT, DB_DEADLOCK, or DB_QUEUE_THR_SUSPENDED, or DB_SUCCESS_LOCKED_REC;
        DB_SUCCESS_LOCKED_REC means that there was a deadlock, but another transaction
        was chosen as a victim,
        and we got the lock immediately: no need to wait then */
dberr_t
RecLock::deadlock_check(lock_t* lock) //检查并解决死锁问题
{...
    trx_mutex_exit(m_trx);
    const trx_t* victim_trx;
    victim_trx = DeadlockChecker::check_and_resolve(lock, m_trx); //挑出一个受害者
    trx_mutex_enter(m_trx);

    /* Check the outcome of the deadlock test. It is possible that
    the transaction that blocked our lock was rolled back and we were granted our lock. */
    dberr_t err = check_deadlock_result(victim_trx, lock);

    ...
    return(err);
}
```

其中，第一步较为复杂，InnoDB 调用 `check_and_resolve()` 函数，通过等待图 (Wait-For-Graph) 的算法来实现死锁检测，即检查持锁者所持有的锁与本事务的新施加的锁之间是否已经成环；如果有环存在说明已经出现死锁（调用 `DeadlockChecker::search()` 函数），根据一定的策略某个事务回滚将环切断而解除死锁。这样的策略是：深度遍历持锁者的锁队列，从中找出受害者；但是如果搜索的深度太深，则不再进行深度搜索而是直接把当前事务或持锁者事务作为受害者，或者选择事务权重<sup>②</sup>小者为受害者（回滚对数据库系统修改少的事务，减少对系统的影响）。

```
/** Checks if a joining lock request results in a deadlock. If a deadlock is found
this function will resolve the deadlock by choosing
a victim transaction and rolling it back. It will attempt to resolve all
deadlocks. The returned transaction id will be the joining
```

① 当然，如第2章所言，如果使用基于时间戳或乐观的并发控制技术，死锁是不可能发生的。

② 事务权重要计算UNDO的量和LOCK的量：

```
#define TRX_WEIGHT(t)((t->undo_no + UT_LIST_GET_LEN((t->lock.trx_locks)))
```



```

transaction instance or NULL if some other transaction was chosen as a victim and
rolled back or no deadlock found.
@param lock lock the transaction is requesting //参数: 准备申请的锁
@param trx transaction requesting the lock //参数: 准备申请锁的事务
@return transaction instance chosen as victim or 0 */
const trx_t*
DeadlockChecker::check_and_resolve(const lock_t* lock, const trx_t* trx)
{...
    /* Try and resolve as many deadlocks as possible. */
    do {
        DeadlockChecker checker(trx, lock, s_lock_mark_counter);
        //通用的死锁检测类, 每轮循环都构造一个新的DeadlockChecker对象
        victim_trx = checker.search(); //从持锁者所在的锁队列中进行深度遍历, 找出一个受害者。
        //search() 内部实现是一个循环, 自身又处于一个循环中, 所以可能很耗时

        /* Search too deep, we rollback the joining transaction only if it is
           possible to rollback. Otherwise we rollback the
           transaction that is holding the lock that the joining transaction wants. */
        if (checker.is_too_deep()) { //如果暂时没有找到, 而又陷入太深的层次(因深度遍历), 则不再寻找, 可能挑选当前事务为受害者
            ut_ad(trx == checker.m_start);
            victim_trx = trx_arbitrate(trx, checker.m_wait_lock->trx);
            //上一节对此函数有分析
            if (victim_trx == NULL) {
                victim_trx = trx; //暂时没有找到受害者挑选当前事务为受害者
            }
            rollback_print(victim_trx, lock);
            MONITOR_INC(MONITOR_DEADLOCK);
            break;
        } else if (victim_trx != 0 && victim_trx != trx) { //找到的受害者不是当前事务, 则
            ut_ad(victim_trx == checker.m_wait_lock->trx);
            checker.trx_rollback(); //把受害者事务回滚掉
            lock_deadlock_found = true;
            MONITOR_INC(MONITOR_DEADLOCK);
        }
    } while (victim_trx != NULL && victim_trx != trx); //循环遍历, 直到找到受害者为止
    ...
}

```

在上面的代码分析中, 通过 `is_too_deep()` 函数判断深度搜索的层次太深则不再进行搜索的条件是:

- 调用层次 `m_n_elems` 的值超过宏 `LOCK_MAX_DEPTH_IN_DEADLOCK_CHECK` 的值 200。
- 搜索节点的个数 `m_cost` 超过宏 `LOCK_MAX_N_STEPS_IN_DEADLOCK_CHECK` 的值 1000000。

## 7. 记录锁的销毁

记录锁，在不再被使用时，通过调用 `lock_rec_discard()` 函数销毁记录锁，这种方式用于各种记录锁，包括 11.6.2 节提及的谓词锁。

可以通过图 11-3 的调用关系，追溯记录锁销毁场景，如调用 `btr_compress()` 函数做页面数据压缩的时候，可能会去掉某个 block，则可以销毁此 block 上的记录锁。

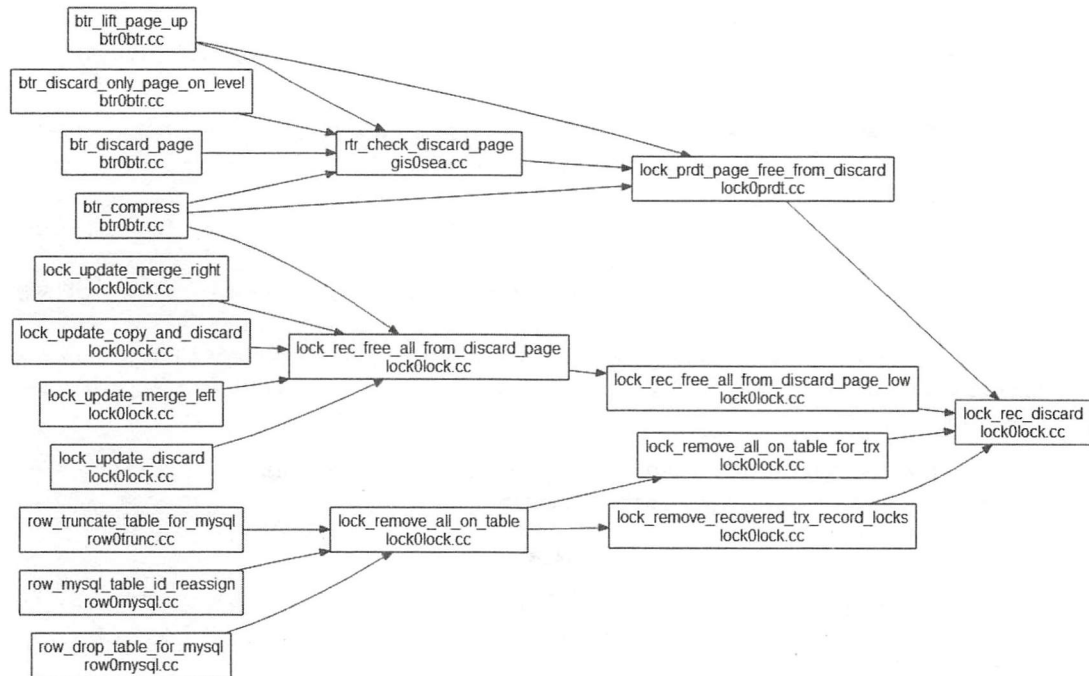


图 11-3 记录锁销毁上下文图

### 11.3.3 记录锁与隔离级别

在 InnoDB 中，记录锁是用于锁定索引上的记录项而非物理元组，但属于对表数据的操作，因此与 DQL 和 DML 有莫大的关系。

在讨论 DQL 和 DML 与锁时，又离不开数据的一致性，所以避免不了对隔离级别的讨论：

- ❑ 可串行化：隔离级别中的可串行化级别从原理上保证了数据的读一致性。
- ❑ 可重复读：InnoDB 的实现方式基于索引组织表，且在可重复读级别在索引上实现了“next key locking”避免了幻象现象，从实践上在可重复读这个级别上保证了数据的一致性。
- ❑ 已提交读和未提交读：已提交读和未提交读被 InnoDB 支持，但是，并不能保证数据的一致性，用户使用的时候，注意自己的场景是否适合这两个级别，如果适合，使用这两个级别时能够提高并发控制系统的并发度，从而提高数据库引擎的执行效率。



## 1. 隔离级别

InnoDB 支持 ANSI SQL 定义的四种隔离级别，内容如下：

```
/* Transaction isolation levels (trx->isolation_level) */
#define TRX_ISO_READ_UNCOMMITTED 0 /* dirty read: non-locking
//脏读，不需要给本事务操作的数据项加任何锁，允许其他事务来读取
    SELECTs are performed so that we do not look at a possible earlier version of
    a record; //查询总是读取最新的数据
    thus they are not 'consistent' reads under this isolation level; otherwise
    like level 2 */

#define TRX_ISO_READ_COMMITTED 1 /* somewhat Oracle-like isolation, //已提交读
    except that in range UPDATE and DELETE we must block phantom rows with next-
    key locks;
    SELECT ... FOR UPDATE and ... LOCK IN SHARE MODE only lock the index records,
    NOT the gaps before them, and thus
    allow free inserting; //已提交读隔离级别不阻止幻象（即允许插入数据），是因为不加间隙锁
    只锁定记录
    each consistent read reads its own snapshot */ //一个事务中的所有读操作（如所有的
    SELECT操作）不使用同一个快照，使用每一个SELECT自己快照，这样有多个不同的快照，所以不能保持读数
    据是一致的

#define TRX_ISO_REPEATABLE_READ 2 /* this is the default;
//可重复读，一个事务中的所有读操作（如所有的SELECT操作），
    all consistent reads in the same trx read the same snapshot;
    //使用同一个快照，所以能保持读数据是一致的
    full next-key locking used in locking reads to block insertions into gaps */
    //使用“next-key locking”阻止其他会话插入数据发生幻象

#define TRX_ISO_SERIALIZABLE 3 /* all plain SELECTs are converted to LOCK
    IN SHARE MODE reads */
```

对于 TRX\_ISO\_READ\_COMMITTED 级别，不施加间隙锁（GAP）；否则隔离级别高于 TRX\_ISO\_READ\_COMMITTED 级别则需要加间隙锁。例如 row\_ins\_duplicate\_error\_in\_clust() 的判断如下：

```
lock_type =
    trx->isolation_level <= TRX_ISO_READ_COMMITTED
    ? LOCK_REC_NOT_GAP : LOCK_ORDINARY;
```

row\_sel\_get\_clust\_rec() 函数中也有类似的判断，如下所示：

```
if (srv_locks_unsafe_for_binlog
    || trx->isolation_level <= TRX_ISO_READ_COMMITTED) {
    lock_type = LOCK_REC_NOT_GAP; //不加间隙锁
} else {
    lock_type = LOCK_ORDINARY; //加间隙锁
}
```

row\_sel() 函数中也有类似的判断。代码参见本函数。

## 2. 可串行化的实现

InnoDB 对于可串行化的实现方式，是通过两种方式实现的。

1) 当 SELECT 语句在一个显式的事务块内，如执行表 11-9 中的编号为 1 的情况，将施加 LOCK\_S 锁，根据表 11-6（记录锁事务锁相容表）可知，LOCK\_S 锁排斥写锁，所以可串行化隔离级别下只允许并发地读取操作，并发写被禁止，因此实现了可串行化（参见 1.2.2 节）。相应代码如下：

```
ha_innobase::external_lock(...)
{...
    if (lock_type != F_UNLOCK) {
        /* MySQL is setting a new table lock */
        ...

        if (trx->isolation_level == TRX_ISO_SERIALIZABLE //可串行化隔离级别
            && m_prebuilt->select_lock_type == LOCK_NONE
            && thd_test_options(thd, OPTION_NOT_AUTOCOMMIT | OPTION_BEGIN)) {
            //且在一个显式事务块内部

            /* To get serializable execution, we let InnoDB conceptually add
            'LOCK IN SHARE MODE' to all SELECTs
            which otherwise would have been consistent reads. An exception is
            consistent reads in the AUTOCOMMIT=1 mode:
            we know that they are read-only transactions, and they can be
            serialized also if performed as consistent reads. */
            m_prebuilt->select_lock_type = LOCK_S; //加读锁，即'LOCK IN SHARE MODE'
            m_prebuilt->stored_select_lock_type = LOCK_S;
        } //否则，不加锁（这一点也很重要）

        ...
    } else {
        TrxInInnoDB::end_stmt(trx);
        DEBUG_SYNC_C("ha_innobase_end_statement");
    }
    ...}
```

2) 当 SELECT 语句不在一个显式的事务块内，则通过获取最新快照（在事务开始的时候，），然后读取数据。此时，因基于快照的一致性读不需要加锁，所以其加锁情况对应到了表 11-9 中编号 2 对应的情况。

表 11-9 可串行化隔离级别加锁情况

隔离级别	编号	SQL 语句	施加的锁
SERIALIZABLE	1	BEGIN; S0;	LOCK_S + LOCK_REC_NOT_GAP
	2	S0;	没有锁

说明：



❑ S0: SELECT \* FROM bluesea WHERE c1=2; // 使用主键索引做 WHERE 条件

另外，对于 FLUSH...WITH READ LOCK 语句，可串行化隔离级别下也需要加读锁 LOCK\_S，代码如下：

```
ha_innbase::store_lock(
...
    /* Check for FLUSH TABLES ... WITH READ LOCK */
    if (trx->isolation_level == TRX_ISO_SERIALIZABLE) {
        m_prebuilt->select_lock_type = LOCK_S;
        m_prebuilt->stored_select_lock_type = LOCK_S;
    } else {
        m_prebuilt->select_lock_type = LOCK_NONE;
        m_prebuilt->stored_select_lock_type = LOCK_NONE;
    }
...
}
```

与可串行化相关的，还有 innbase\_query\_caching\_of\_table\_permitted() 函数，可串行化隔离级别不允许缓冲查询。

### 3. 可重复读的实现

可重复读隔离级别，简称 RR，在 2.1.1 节，我们说过：

**Repeatable Read (可重复读)：**一个事务在执行过程中可以看到其他事务已经提交的新插入的记录（读已经提交的，其实是读早于本事务开始且已经提交的），但是不能看到其他事务对已有记录的更新（即晚于本事务开始的），并且，该事务不要求与其他事务是“可串行化”的。

这句话的核心，是“但是不能看到其他事务对已有记录的更新”，那么 RR 隔离级别是怎么保证这点的呢？

如果数据库并发控制引擎是单纯的封锁协议机制，则应该在读取数据的时候，判断数据项是不是其他事务更新过的。可是 InnoDB 没有这么做，而是通过如下方式，在 RR 隔离级别下为事务设置了一个“一致性读视图（即快照）”，之后读取数据，就是根据这个快照来获取，这样，就不能看到他晚于本事务的事务对已有记录的更新（更新生成新版本，必然不在旧的快照所限定的范围内）。

```
static my_bool snapshot_handleron(THD *thd, plugin_ref plugin, void *arg)
{
    handleron *hton= plugin_data<handleron*>(plugin);
    if (hton->state == SHOW_OPTION_YES && hton->start_consistent_snapshot)
        // 隔离级别是RR时start_consistent_snapshot才被赋值
    {
        hton->start_consistent_snapshot(hton, thd);
        //对于InnoDB, 实际执行innbase_start_trx_and_assign_read_view()函数
    }
}
```

```

        *((bool *)arg)= false;
    }
    return FALSE;
}

```

如图 11-4 所示，以及下面的代码分析，在事务开始的时候 `trans_begin()` 会调用 `snapshot_handleron()` 函数指针即使用 `innobase_start_trx_and_assign_read_view()` 函数在可重复读隔离级别下创建一个快照，其他隔离级别则不创建快照。

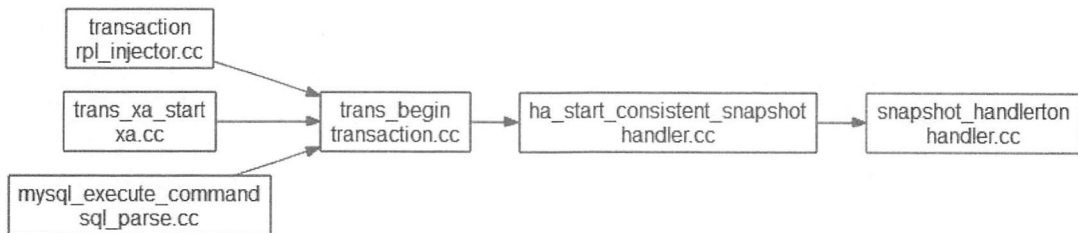


图 11-4 snapshot\_handleron() 函数上下文调用关系

```

innobase_start_trx_and_assign_read_view(
//在可重复读隔离级别下创建一个快照，其他隔离级别则不创建快照
    handlerton*      hton,      /*!< in: InnoDB handlerton */
    THD*             thd)       /*!< in: MySQL thread handle of the user for whom the
                                transaction should be committed */
{
    ...
    if (trx->isolation_level == TRX_ISO_REPEATABLE_READ) {
        //如果是RR隔离级别，则给read view赋值，即构建一致性视图
        trx_assign_read_view(trx);
        //为读一致性视图（快照）赋一个值，注意在store_lock()中应隔离级别小于RR才关闭快照
    } else {
        push_warning_printf(thd, Sql_condition::SL_WARNING,
                            HA_ERR_UNSUPPORTED,
                            "InnoDB: WITH CONSISTENT SNAPSHOT"
                            " was ignored because this phrase"
                            " can only be used with"
                            " REPEATABLE READ isolation level.");
    }
    ...
}

```

之后，在每条 SQL 语句执行的时候，根据隔离级别判断是不是要使用一个新的快照，如果是可重复读，则不使用新快照，沿用老的快照，这样就能保证所有的读操作看到的是同一个数据状态；同时也确保了已提交读隔离级别下一个事务块内的不同语句的读操作看到的不是同一个数据状态。

```

ha_innobase::store_lock(...)

```



```

{...
    if (lock_type != TL_IGNORE && trx->n_mysql_tables_in_use == 0) {
        trx->isolation_level = innobase_map_isolation_level((enum tx_isolation) thd_tx_isolation(thd));
        if (trx->isolation_level <= TRX_ISO_READ_COMMITTED
            // 隔离级别小于等于已提交读，关闭老的快照。可重复读不关闭老快照所以可以沿用
            && MVCC::is_view_active(trx->read_view)) {
            /* At low transaction isolation levels we let each consistent read
               set its own snapshot */
            mutex_enter(&trx_sys->mutex);
            trx_sys->mvcc->view_close(trx->read_view, true);
            // 隔离级别小，关闭快照，这样下一条SQL执行时，将获取新快照
            mutex_exit(&trx_sys->mutex);
        }
    }
    ...
}

```

从上面的分析可以看出，InnoDB的可重复读的实现，利用了实现MVCC技术的快照技术。这是MVCC和基于封锁技术这两个并非控制技术的结合之处。

#### 4. 已提交读的实现

对于已提交读隔离级别的实现方式，从逻辑上需要明确两个部分，一是加锁部分，二是解锁部分。加锁，对应的是获取数据，确保在指定的隔离级别下读取到应该读到的数据。解锁则意味着要在适当的时机释放锁且不影响隔离级别的语义还能提高并发度。

加锁部分，实现分为两个方面：一是加锁的时候，已提交读隔离级别不加间隙锁，这样就能允许并发的其他事务执行插入操作因而产生幻象现象，因为已提交读隔离级别是允许幻象异常存在的。如下代码所示，加锁的时候，根据隔离级别判断是否加间隙锁。

```

row_sel_get_clust_rec⊖(...)
{...
    if (!node->read_view) {
        ...
        if (srv_locks_unsafe_for_binlog
            || trx->isolation_level <= TRX_ISO_READ_COMMITTED) {
            lock_type = LOCK_REC_NOT_GAP;
            // 小于等于已提交读，则不加间隙锁，允许其他事务插入，因此可发生幻象
        } else {
            lock_type = LOCK_ORDINARY;
            // 大于已提交读，则加间隙锁，防止其他事务插入某个范围内的数据，避免幻象
        }
    }
    ...}

```

其次，要确定可以读取到什么样的元组，即判断是不是没有被提交的元组也可以读到。

⊖ 位于row0sel.cc文件中。

既然是读已提交级别，则必然是只能读取到已经被提交的元组，这样才能体现“已提交”的含义。这时，就涉及数据的可见性判断的问题（本节不讨论可见性问题，详情参见12.2节）。

解锁部分，要及时释放锁，这样便于其他事务能够读取到不应当被本事务锁定的记录（InnoDB中“记录”是索引项，通过记录才能真正找到元组）。以索引上的范围扫描为例，查看锁的释放条件。

```
ha_innpart::read_range_next_in_part(...)
//Return next record in index range scan from a partition
{...
    error = ha_innobase::index_next(read_record);
    //获得记录，则会加锁，此时error的值被赋予0
    if (error == 0 && !in_range_check_pushed_down) { //记录被加过了锁
        /* compare_key uses table->record[0], so we need to copy the data if not already there. */
        if (record != NULL) {
            copy_cached_row(table->record[0]⊖, read_record);
            //复制获取到的元组到表级的数据缓冲区
        }
        if (compare_key(end_range) > 0) { //超出要读取的范围，则释放锁
            /* must use ha_innobase:: due to set/update_partition
            could overwrite states if ha_innpart::unlock_row() was used. */
            ha_innobase::unlock_row(); //释放锁
            error = HA_ERR_END_OF_FILE;
        }
    }
    ...
}
```

根据隔离级别确定是否要释放锁。

```
/** Removes a new lock set on a row, if it was not read optimistically. This can
    be called after a row has been read
    in the processing of an UPDATE or a DELETE query, if the option innodb_locks_
    unsafe_for_binlog is set. */
void //被mysql_update()/mysql_delete()调用，用于为记录解锁。另外少数情况是：被join_
read_key()等调用
ha_innobase::unlock_row(void) //在UPDATE或DELETE执行时，一个元组被读取操作后，所施加
的锁“可能”被本方法释放
{... //所施加的锁是否被释放，取决于下面对隔离级别的判断
    switch (m_prebuilt->row_read_type) {
    case ROW_READ_WITH_LOCKS:
        if (!srv_locks_unsafe_for_binlog
            && m_prebuilt->trx->isolation_level
            > TRX_ISO_READ_COMMITTED) {
            //隔离级别是可重复读或可串行化，则满足大于已提交读，所以执行break不解锁
        }
    }
}
```

⊖ 执行器使用的表的数据就是从table->record[0]获得的。



```

        break;
    }
    /* fall through */
case ROW_READ_TRY_SEMI_CONSISTENT:
    row_unlock_for_mysql(m_prebuilt, FALSE);
    //如果是已提交读隔离级别, 则能执行到解锁操作
    break; //意味着已提交读隔离级别加锁过后, 则释放锁, 而不是等待事务结束时释放锁。所以
           更新等操作可以被其他事务有机会看到⊖
case ROW_READ_DID_SEMI_CONSISTENT:
    m_prebuilt->row_read_type = ROW_READ_TRY_SEMI_CONSISTENT;
    break;
}
...
}

```

紧接着, 判断并发事务间的提交关系 (涉及了可见性判断规则: 通过 `lock_clust_rec_cons_read_sees()` 调用 `changes_visible()` 利用元组上的事务 ID 与快照的左右边界比较), 然后再确定是否是解锁。如下是解锁的过程。

```

/** This can only be used when srv_locks_unsafe_for_binlog is TRUE or this
session is using a READ COMMITTED or READ UNCOMMITTED isolation level.
Before calling this function row_search_for_mysql() must have initialized
prebuilt->new_rec_locks to store the information which new
record locks really were set. This function removes a newly set clustered index
record lock under prebuilt->pcur or
prebuilt->clust_pcur. Thus, this implements a 'mini-rollback' that releases the
latest clustered index record lock we set.
@param[in,out]    prebuilt          prebuilt struct in MySQL handle
@param[in]        has_latches_on_recs TRUE if called so that we have the
latches on the records under pcur
                                and clust_pcur, and we do not need
to reposition the cursors. */
void
row_unlock_for_mysql(row_prebuilt_t* prebuilt, ibool has_latches_on_recs)
{...
    if (prebuilt->new_rec_locks >= 1) {
    ...
        /* If the record has been modified by this transaction, do not unlock it. */
        if (index->trx_id_offset) {
            //如果是被本事务修改, 则不释放锁 (修改元组则会写事务ID到元组中)
            rec_trx_id = trx_read_trx_id(rec + index->trx_id_offset);
            //获得元组上的事务id值
        } else {...
            offsets = rec_get_offsets(rec, index, offsets, ULINT_UNDEFINED, &heap);
            rec_trx_id = row_get_rec_trx_id(rec, index, offsets);
        }
    }
}

```

⊖ 注意, 只是存在能被其他事务读到修改后的数据的可能, 单是还没有判断事务是否已经提交。

```

//获得元组上的事务id值
    if (UNIV_LIKELY_NULL(heap)) {
        mem_heap_free(heap);
    }
}

if (rec_trx_id != trx->id) {
    //元组上的事务id不是本事务的id, 表明元组是被其他事务修改, 释放锁
    /* We did not update the record: unlock it */
    rec = btr_pcur_get_rec(pcur);
    lock_rec_unlock(trx, btr_pcur_get_block(pcur), rec, static_cast<enum
lock_mode>(prebuilt->select_lock_type));

    if (prebuilt->new_rec_locks >= 2) { //new_rec_lock通常是0, 如果隔离级别
是READ COMMITTED或READ UNCOMMITTED
        rec = btr_pcur_get_rec(clust_pcur); //则在row_search_mvcc()中获得
记录锁后设置为2, 所以需要对应解锁
        lock_rec_unlock(trx, btr_pcur_get_block(clust_pcur), rec, static_
cast<enum lock_mode>(prebuilt->select_lock_type));
    }
}

no_unlock:
    mtr_commit(&mtr);
}
...
}

```

在一个事务块内, 如果存在多条 SELECT 语句, 则在已提交读隔离级别下, 每条 SELECT 语句分别使用自己的快照 (Read view, 即为每条 SELECT 生成一个 Read view, 每条 SELECT 结束后, 通过调用 MVCC::view\_close() 方法, Read view 会被关闭)。

对于一个 UPDATE 或 DELETE 操作, 当有页面 (索引页面) 因增加或删除了元组而分离或合并时, 需要让新页继承旧页的锁信息, 这时继承操作是通过 lock\_rec\_add\_to\_queue() 函数加锁完成的, 但是, 加锁时会有间隙锁存在, 代码如下:

```

lock_rec_add_to_queue( //被lock_rec_inherit_to_gap()调用, 在原先的锁的基础上加持间隙锁GAP
    LOCK_REC | LOCK_GAP | lock_get_mode(lock),
    //lock_get_mode(lock)是原先锁的粒度和类型, LOCK_GAP是必须加持的类型
    heir_block, heir_heap_no, lock->index,
    lock->trx, FALSE);

```

## 5. 未提交读的实现

对于未提交读隔离级别, 此级别不会对记录加锁, 有如下几种情况:

- 对系统表的数据操作, 是数据引擎自己发出的数据查询操作, 使用未提交读隔离级别, 目的是不与其他事务因锁的存在而冲突。



□ 在 `row_search_mvcc()`、`row_sel_get_clust_rec_for_mysql()` 等获取记录的函数中确保读未提交隔离级别下允许读到最新的记录。

那么，怎么才能读到最新的记录呢？以 `row_search_mvcc()` 为例，我们来看代码实现方式：对于未提交读的隔离级别，代码什么也没有做，默认本函数的“PHASE 4”调用“`rec = btr_pcur_get_rec(pcur);`”获得的 `rec` 就是可用的。如果不是未提交读的隔离级别，才判断 `rec` 的可见性，`rec` 不可见时才去找旧版本。所以，InnoDB 以此方式，实现了未提交读隔离级别。与此方式类似的处理，还有 `row_sel_get_clust_rec_for_mysql()` 函数，读者可以参考阅读。

```
row_search_mvcc(...) //获取记录
{...
    /* We are ready to look at a possible new index entry in the result
       set: the cursor is now placed on a user record */
    if (prebuilt->select_lock_type != LOCK_NONE) { //如果需要加锁，则加锁的语义来自上层
        ...
    } else { //不需要加锁
        /* This is a non-locking consistent read: if necessary, fetch a previous
           version of the record */
        if (trx->isolation_level == TRX_ISO_READ_UNCOMMITTED) {
            //如果是未提交读，则什么也不做
            /* Do nothing: we let a non-locking SELECT read the latest version of the record */
        } else if (index == clust_index) { //否则，才需要检查“rec”是否满足可见性，不
            满足则找出满足可见性要求的旧版本
            //所以，未提交读什么也不做，就是在说当前的rec是符合的，能被读到的，根本不用到回滚
            段找出旧版本（所得即所“可见”）
            /* Fetch a previous version of the row if the current one is not
               visible in the snapshot;
               if we have a very high force recovery level set, we try to avoid
               crashes by skipping this lookup */
            if (srv_force_recovery < 5
                && !lock_clust_rec_cons_read_sees(rec, index, offsets, trx_get_read_view(trx))) {
                rec_t* old_vers;
                /* The following call returns 'offsets' associated with 'old_vers' */
                err = row_sel_build_prev_vers_for_mysql(trx->read_view, clust_
                    index, prebuilt, rec, &offsets, &heap,
                        &old_vers, need_vrow ?
                            &vrow : NULL, &mtr);
                if (err != DB_SUCCESS) {
                    goto lock_wait_or_error;
                }
                if (old_vers == NULL) {
                    /* The row did not exist yet in the read view */
                    goto next_rec;
                }
                rec = old_vers;
            }
        }
    }
}
```

```

    }
} else {
    /* We are looking into a non-clustered index, and to get the right
       version of the record we
       have to look also into the clustered index: this is necessary,
       because we can only get the undo
       information via the clustered index record. */
    ...
}
}
}
...}

```

## 11.4 事务锁之元数据锁

对于一个表对象而言，并发操作，可能修改表的元数据、也可能修改表中的用户数据，所以这些内容都需要进行保护。之前讨论的记录锁，就是防止并发修改表中的用户数据出现数据不一致的现象。而元数据锁，指的是并发修改表的元数据时需要加的锁。但是，数据库中对象很多，不仅是表对象还有其他诸如视图、存储过程、表空间等对象，并发修改这些对象，也存在数据的一致性问题，只是这里的数据指的是“元数据”而不是“用户数据”所以，MySQL 统一使用 MDL 锁 (metadata lock) 来表示“元数据”上的锁。

这段话，其实也在表明，为什么数据库需要锁（并发情况下保护共享资源）；还在表明，为什么数据库有不同对象的锁（并发情况下保护不同的对象）。

MySQL 的元数据锁，首先体现在 MySQL Server 层，这是本节讨论的重点内容。

元数据锁在整个 MySQL 体系中，其重要程度，不亚于记录锁。这是因为元数据锁涉及的范围更广：从 SQL 语句的角度看，除了与记录锁有关的 DML、DQL 语句外，DDL 语句也与元数据锁紧密相关；从数据库内部的对象看，除了表对象与元数据锁相关外，还有表空间、存储过程、用户自定义函数等对象涉及“元数据”锁，对于这些对象而言，准确的说法就是“元数据锁”，所以对于表对象不再称为“表锁”而是统一到了“元数据锁”这个概念中。

下面我们在 MySQL Server 层和 InnoDB 层互相穿插，讨论元数据锁的实现（元数据锁的相关代码，主要位于 MySQL Server 层，更为深入的内容参见 11.5 节）。

有关 MDL 锁的发展历史，可以参考：<http://mysqllover.com/?p=985>。

### 11.4.1 元数据锁的数据结构

接下来，先从重要的数据结构和概念上来认识元数据锁。

#### 1. InnoDB 定义的表锁

InnoDB 层定义了一个数据结构，用于关联 InnoDB 层内表对象和锁之间的关系。





```

/** A table lock */
struct lock_table_t { //表锁对象, 在InnoDB层内表示表对象和锁之间的关系
    dict_table_t* table; /*!< database table in dictionarycache */
    //在数据字典cache中的表对象(元数据), 处于InnoDB层
    UT_LIST_NODE_T(lock_t) locks; /*!< list of locks on the sametable */
    //同一个表上的所有锁的list
};

```

## 2. 元数据锁的类别

MySQL Server 层, 定义了全局的元数据锁, 而元数据被分为几种类型, 这一点可以从枚举类型“enum\_mdl\_namespace”中看出, 划分的依据是根据被加锁的对象的类别区分的, 这些被加锁的对象是数据库引擎所包含的对象, 内容如下:

```

enum enum_mdl_namespace { //锁的命名空间, 表明锁的作用范围, 实际上是锁作用的对象是什么
    GLOBAL=0, //全局锁, 当需要为系统重新加载权限信息和缓冲、或标识系统为只读模式时使用, 参见
        reload_acl_and_cache()
        //其他使用方式可见下面内容, 需要和TABLESPACE、SCHEMA等配合使用, 以示“全局”之意
        //因为有些操作是全局且在同一个对象禁止并发地, 如创建一个数据库、删除一个数据库、
        升级一个数据库
        //这种情况下, 一个语句是一个事务。所以如lock_schema_name()/lock_tablespace_name()中
        //先为GLOBAL赋予MDL_STATEMENT, 接着为SCHEMA/TABLESPACE赋予MDL_
        TRANSACTION表示锁的生命周期
        //GLOBAL并不对应DDL, 可参考get_deadlock_weight()函数中的注释。如open_
        table()调用acquire_lock()使用了GLOBAL
    TABLESPACE, //表空间类型的锁
    SCHEMA, //表之上的对象(database)类型的锁, 当操作一个“数据库实例中的一个数据库
        时”, 需要加GLOBAL和SCHEMA类型的锁
    TABLE, //表类型的锁
    FUNCTION, //函数(用户自定义函数)类型的锁
    PROCEDURE, //存储过程类型的锁
    TRIGGER, //触发器类型的锁
    EVENT, //事件调度类型的锁, 用于“CREATE EVENT...”等于事件相关的操作中。事件是一
        个观察点, 可实施作业调度等工作
    COMMIT, //提交保护锁, 允许全局的读锁阻塞提交操作, 如“FLUSH TABLES WITH READ
        LOCK⊖”中的读锁是全局读锁
    USER_LEVEL_LOCK, //通过GET_LOCK()/RELEASE_LOCK()函数操作施加的锁
    LOCKING_SERVICE, //加锁接口锁。加锁接口为MySQL数据库引擎对外提供的一套接口, 供用户实现
        自己的加锁、解锁功能
    /* This should be the last ! */
    NAMESPACE_END };

```

枚举类型“enum\_mdl\_namespace”被定义在结构体“struct MDL\_key”中, 此结构体表示某一类元数据锁, 被视为某一类元数据锁的“key”标识。其内容如下:

⊖ 如刷出数据的时候, 禁止事务提交, 保证了数据的一致性。



```

struct MDL_key    //Metadata lock object key.
{
    ...
    enum enum_mdl_namespace {...}; //本小节开始处定义的元数据锁的类型

    const uchar *ptr() const { return (uchar*) m_ptr; }
    //为MDL_key提供的一些相关操作，如下还有，省略...
    uint length() const { return m_length; }

    const char *db_name() const { return m_ptr + 1; }
    uint db_name_length() const { return m_db_name_length; }

    ...
};

```

每一类锁，持有的时间是由确定范围的，这个范围，是锁的生命周期，使用 `enum_mdl_duration` 表示：

```

/** Duration of metadata lock. */
enum enum_mdl_duration { //锁的生命周期
    /**Locks with statement duration are automatically released at the end of
    statement or transaction.*/
    MDL_STATEMENT= 0, //语句级，语句或事务执行结束锁被自动释放。少数操作是语句级的，如涉及
    SCHEMA/TABLESPACE（上面有说明）
    /**Locks with transaction duration are automatically released at the end of transaction.*/
    MDL_TRANSACTION, //事务级，事务执行结束锁被自动释放。多数操作是事务级的，如对表的各种各样的操作
    /**Locks with explicit duration survive the end of statement and transaction.
    They have to be released explicitly by calling MDL_context::release_lock(). */
    MDL_EXPLICIT, //显式级，锁被系统使用MDL_context::release_lock()释放，如LOCK TABLE、
    FLUSH .. WITH LOCK
    /* This should be the last ! */
    MDL_DURATION_END };

```

锁的命名空间“`enum_mdl_namespace`”中所定义的各种类型的锁，在 `MDL_lock` 中被划分为两大类：

- 一类是 `scoped` 类型，包括锁的命名空间“`enum_mdl_namespace`”中所定义的 `GLOBAL`、`COMMIT`、`TABLESPACE` 和 `SCHEMA`。
- 另外一类是 `object` 类型，包括锁的命名空间“`enum_mdl_namespace`”中所定义其他类型。

### 3. 元数据锁的粒度

MySQL Server 层定义了如下的元数据锁的粒度，这些锁较多，超出了读锁和写锁两种锁粒度，这是因为读锁和写锁通常是理论上根据概念探讨的范畴，而在工程实践中会根据实际情况进行细化，以最大限度地提高并发度。

```

/**
    Type of metadata lock request.

```





```

    @sa Comments for MDL_object_lock::can_grant_lock() and MDL_scoped_lock::can_
    grant_lock() for details.
*/
enum enum_mdll_type { //元数据锁的粒度定义, 但属于MySQL Server层的元数据锁。如下的注释写得很好, 仔细阅读理解为好
    /*An intention exclusive metadata lock. Used only for scoped locks.
    //scoped locks, 范围类型的锁, 参见“本节11.相容性”小节
    Owner of this type of lock can acquire upgradable exclusive locks
    on individual objects. //可升级的排它锁, 从意向升级到非意向的排它
    Compatible with other IX locks, but is incompatible with scoped S and X locks.*/
    MDL_INTENTION_EXCLUSIVE= 0, //意向排它锁, 缩写为IX。与其他的IX是兼容的, 但是与范围类型的S
    和X锁不兼容

                                //此锁用于GLOBAL/COMMIT/TABLESPACE/SCHEMA/EVENT/
                                PROCEDURE/TABLE这些元数据对象上

    /*A shared metadata lock. //共享锁, 元数据共享锁, 即表级的共享锁
    To be used in cases when we are interested in object metadata only and
    there is no intention to //读操作“元数据”获取S锁
    access object data (e.g. for stored routines or during preparing prepared
    statements). //例如存储过程等
    We also mis-use this type of lock for open HANDLERS, since lock acquired
    by this statement has to be compatible with
    lock acquired by LOCK TABLES ... WRITE statement, i.e.//
    “HANDLER tbl_name OPEN/READ...”操作不能获得S锁
    SNRW (We can't get by acquiring S lock at HANDLER ... OPEN time and
    upgrading it to SRlock for HANDLER ... READ
    as it doesn't solve problem with need to abort DML statements which wait
    on table level lock while having
    open HANDLER in the same connection).
    To avoid deadlock which may occur when SNRW lock is being upgraded to X
    lock for table //升级到X锁避免死锁
    on which there is an active S lock which is owned by thread which waits in
    its turn for table-level lock owned by thread
    performing upgrade we have to use thr_abort_locks_for_thread() facility in such situation.
    This problem does not arise for locks on stored routines as we don't use
    SNRW locks for them.
    It also does not arise when S locks are used during PREPARE calls as
    table-level locks are not acquired in this case.*/
    MDL_SHARED, //共享锁, 缩写为S。需要对元数据进行读操作

    /*A high priority shared metadata lock. //高优先级的共享锁, 高优先级的元数据共享
    锁, 即高优先级的S锁
    Used for cases when there is no intention to access object data (i.e. data
    in the table). //用于不需要意向锁的情况下
    “High priority” means that, unlike other shared locks, it is
    granted ignoring pending requests for exclusive locks.

```





```

Intended for use incases when we only need to access metadata and not
data, e.g. whenfilling an INFORMATION_SCHEMA table.
//如上是在说, 高优先级是在在表级操作但不会影响数据 (如修改列的数据类型则会影响数据, 修
改表名则不会影响数据)
Since SH lock is compatible with SNRW lock, the connection thatholds SH
lock lock should not try to
acquire any kind of table-levelor row-level lock, as this can lead to a deadlock.
//持有SH锁则不应该在请求表级或行级锁
Moreover, afteracquiring SH lock, the connection should not wait for any otherresource,
as it might cause starvation for X locks and a potentialdeadlock during
upgrade of SNW or SNRW to X lock
(e.g. if theupgrading connection holds the resource that is being waited for).*/
MDL_SHARED_HIGH_PRIO, //高优先级的共享锁, 缩写为SH。用于操作INFORMATION_SCHEMA中的表, 如
执行SHOW CREATE TABLE t1或DESC t1等

/*A shared metadata lock for cases when there is an intention to read
datafrom table.//存在读数据意向的锁
A connection holding this kind of lock can read table metadata and read
table data//先在元数据上加上此锁
(after acquiring appropriate table and row-level locks).//之后才会获取表锁和行级锁
This means that one can only acquire TL_READ, TL_READ_NO_INSERT,
andsimilar table-level locks on table
if one holds SR MDL lock on it.
To be used for tables in SELECTs, subqueries, and LOCK TABLE ... READstatements.*/
MDL_SHARED_READ, //意向读锁, 缩写为SR。也许会读表的元数据 (表级元数据锁) 和表的数据 (行级锁)
//另外还有通过mysql_admin_table()函数访问表的ANALYZE/CHECK TABLE/OPTIMIZE
TABLE/REPAIR TABLE等操作可获取此锁

/*A shared metadata lock for cases when there is an intention to
modify(and not just read) data in the table.
A connection holding SW lock can read table metadata and modify or
readtable data//存在修改数据意向的锁
(after acquiring appropriate table and row-level locks).//之后才会获取表锁和行级锁
To be used for tables to be modified by INSERT, UPDATE, DELETEstatements,
but not LOCK TABLE ... WRITE or DDL).
Also taken bySELECT ... FOR UPDATE.*/
MDL_SHARED_WRITE, //意向写锁, 缩写为SW。INSERT, UPDATE, DELETE statements

/*A version of MDL_SHARED_WRITE lock which has lower priority thanMDL_
SHARED_READ_ONLY locks.
Used by DML statements modifyingtables and using the LOW_PRIORITY
clause.*/ //SQL语句中带有“LOW_PRIORITY”子句
MDL_SHARED_WRITE_LOW_PRIO, //低优先级的共享写锁, 缩写为SWLP。

/*An upgradable shared metadata lock which allows concurrent updates and
reads of table data.//允许并发更新和读同一个表

```





```

A connection holding this kind of lock can read table metadata and readtable data.
It should not modify data as this lock is compatible withSRO locks.
Can be upgraded to SNW, SNRW and X locks.
Once SU lock is upgraded to X or SNRW lock data modification can happen freely.
To be used for the first phase of ALTER TABLE.*/
MDL_SHARED_UPGRADABLE, //共享可升级锁, 缩写为SU。ALTER TABLE命令早期阶段使用本锁

/* A shared metadata lock for cases when we need to read data from
tableand block all concurrent modifications to it
(for both data and metadata).Used by LOCK TABLES READ statement.*/
MDL_SHARED_READ_ONLY, //共享只读锁, 缩写为SRO。阻塞所有的并发修改

/* An upgradable shared metadata lock which blocks all attempts to
updatetable data, allowing reads.//阻塞锁更新表数据的操作
A connection holding this kind of lock can read table metadata and
readtable data.//允许读元数据和表数据
Can be upgraded to X metadata lock.
Note, that since this type of lock is not compatible with SNRW or SWlock types,
acquiring appropriate engine-level locks for reading(TL_READ* for MyISAM,
shared row locks in InnoDB) should be contention-free.
To be used for the first phase of ALTER TABLE, when copying data between tables,
to allow concurrent SELECTs from the table, but not UPDATES.*/
MDL_SHARED_NO_WRITE, //共享非写锁, 缩写为SNW。

/* An upgradable shared metadata lock which allows other connectionsto
access table metadata, but not data.
It blocks all attempts to read or update table data, while allowing
INFORMATION_SCHEMA and SHOW queries.
//允许其他会话访问表的元数据。但不允许访问表的数据
A connection holding this kind of lock can read table metadata modify
andread table data.//持锁者可读表的元数据和表数据
Can be upgraded to X metadata lock.
To be used for LOCK TABLESWRITE statement.
Not compatible with any other lock type except S and SH. */
MDL_SHARED_NO_READ_WRITE, //共享非读写锁, 缩写为SNRW。用于LOCK TABLES...WRITE

/* An exclusive metadata lock.
A connection holding this lock can modify both table's metadata and data.
No other type of metadata lock can be granted while this lock is held.
To be used for CREATE/DROP/RENAME TABLE statements and for execution
ofcertain phases of other DDL statements. */
MDL_EXCLUSIVE, //排它锁, 缩写为X。ALTER TABLE等一些操作也要用到此锁, 如本地更新时调用
mysql_inplace_alter_table()函数

/* This should be the last !!! */
MDL_TYPE_END};

```



另外，这些枚举元素的顺序需要得到遵守，不能任意调整次序，在死锁检测时需要根据他们的值来决定某类锁的权重值，这一点可参见 `get_deadlock_weight()` 函数。

表 11-10 元数据锁表

锁的类型	缩写	应对的 SQL 操作或对象
MDL_INTENTION_EXCLUSIVE	IX	GLOBAL 对象、SCHEMA 对象操作会加此锁
MDL_SHARED	S	FLUSH TABLES with READ LOCK LOCK TABLES ... WRITE
MDL_SHARED_HIGH_PRIO	SH	仅对 MyISAM 存储引擎有效
MDL_SHARED_READ	SR	SELECTs, subqueries, and LOCK TABLE ... READ
MDL_SHARED_WRITE	SW	DML 语句, INSERT, UPDATE, DELETE
MDL_SHARED_WRITE_LOW_PRIO	SWLP	仅对 MyISAM 存储引擎有效
MDL_SHARED_UPGRADABLE	SU	the first phase of ALTER TABLE
MDL_SHARED_READ_ONLY	SRO	LOCK TABLES xxx READ
MDL_SHARED_NO_WRITE	SNW	FLUSH TABLES xxx,yyy,zzz READ the first phase of ALTER TABLE
MDL_SHARED_NO_READ_WRITE	SNRW	FLUSH TABLE xxx WRITE LOCK TABLES
MDL_EXCLUSIVE	X	ALTER TABLE xxx PARTITION BY ... CREATE/DROP/RENAME TABLE

在 MySQL Server 层，也对元数据锁定义了多种粒度相关的标志，内容如下：

```
/* mysql_lock_tables() and open_table() flags bits */
#define MYSQL_OPEN_IGNORE_GLOBAL_READ_LOCK 0x0001
//用于复制功能避免在备机施加全局读锁（备机不进行死锁判断等），参考open_table()
#define MYSQL_OPEN_IGNORE_FLUSH 0x0002 //忽略FLUSH操作引发的锁，如open_log_table()函数中预先为flags设置此锁，传入open_ltable()
/* MYSQL_OPEN_TEMPORARY_ONLY (0x0004) is not used anymore. */
#define MYSQL_LOCK_IGNORE_GLOBAL_READ_ONLY 0x0008 //忽略只读操作的锁，类似MYSQL_OPEN_IGNORE_GLOBAL_READ_LOCK
#define MYSQL_LOCK_LOG_TABLE 0x0010//特殊的“日志表”锁，如慢日志类型被视为“表”，实则不是表，但同样的使用了元数据锁机制
#define MYSQL_OPEN_HAS_MDL_LOCK 0x0020//Do not try to acquire a metadata lock on the table: we already have one.已经拥有了MDL锁

/**If in locked tables mode, ignore the locked tables and get a new instance of the table.*/
#define MYSQL_OPEN_GET_NEW_TABLE 0x0040
/* 0x0080 used to be MYSQL_OPEN_SKIP_TEMPORARY */
/** Fail instead of waiting when conflicting metadata lock is discovered. */
#define MYSQL_OPEN_FAIL_ON_MDL_CONFLICT 0x0100 //施加MDL锁冲突，则不等待而是报告错误
/** Open tables using MDL_SHARED lock instead of one specified in parser. */
#define MYSQL_OPEN_FORCE_SHARED_MDL 0x0200 //与MDL_SHARED粒度的锁配合表示PREPARE语句在表对象上施加的共享的元数据锁
```





```

/**Open tables using MDL_SHARED_HIGH_PRIO lock instead of one specified in parser.*/
#define MYSQL_OPEN_FORCE_SHARED_HIGH_PRIO_MDL 0x0400 //与上一个锁类似,只是用于MDL_SHARED_HIGH_PRIO
/**When opening or locking the table, use the maximum timeout
    (LONG_TIMEOUT = 1 year) rather than the user-supplied timeout value.*/
#define MYSQL_LOCK_IGNORE_TIMEOUT 0x0800 //在某些特定的表(如mysql.db、mysql.user、mysql.event)上操作,忽略超时机制
/**When acquiring "strong" (SNW, SNRW, X) metadata locks on tables to
    be open do not acquire global, tablespace-scope and schema-scope IX locks.*/
#define MYSQL_OPEN_SKIP_SCOPED_MDL_LOCK 0x1000
/**When opening or locking a replication table through an internal operation
    rather than explicitly through an user thread.*/
#define MYSQL_LOCK_RPL_INFO_TABLE 0x2000 //仅用于复制功能
/**Only check THD::killed if waits happen (e.g. wait on MDL, wait on
    table flush, wait on thr_lock.c locks) while opening and locking table.*/
#define MYSQL_OPEN_IGNORE_KILLED 0x4000

```

MySQL Server 层的这些锁(锁标志)将被传递到底层的存储引擎中,InnoDB 作为存储引擎之一,对于元数据级操作接受上层施加的这些元数据锁,对于元组级操作根据 SQL (DQL、DML) 语义施加元组锁(参见 11.3 节),从而完成不同的 SQL 操作语义。

#### 4. 元数据锁的锁请求与锁等待

元数据锁在 MySQL Server 层,按照锁的状态被细分为两种,一种是已经施加的锁,一种是等待施加的锁即锁请求,这样被区分的原因,如 MySQL 对“class MDL\_request”的代码注释作了解释:

```

/**
    A pending metadata lock request.

    A lock request and a granted metadata lock are represented by
    different classes because they have different allocation
    sites and hence different lifetimes. The allocation of lock requests is
    //注意这里给出的原因是从工程实践中对锁的实现的角度区分的
    controlled from outside of the MDL subsystem, while allocation of granted
    //体现了工程与理论的不同
    locks (tickets) is controlled within the MDL subsystem.

    MDL_request is a C structure, you don't need to call a constructor
    or destructor for it.
*/
class MDL_request{...} //锁请求

```

所以,元数据锁在使用的过程中又被细分为“lock granted metadata”(代码中使用“class MDL\_ticket”表示.ticket,入场卷,即要被授予的锁)和“lock request metadata”(代码中使用“class MDL\_request”表示加锁的请求)。这样,MySQL Server 分别使用两个类



来表示这两种被细分的锁。

当锁请求要求施加锁不成功的时候，则产生锁等待，而锁等待则用 MDL\_wait 表示。

```
/**A reliable way to wait on an MDL lock. */
class MDL_wait{... //锁等待。在获取锁的过程中，需要为是否获得锁做标志，采用的就是锁等待的状态值
enum enum_wait_status { //锁等待的状态，在锁等待在生命周期内因不同操作产生不同结果
    EMPTY = 0, //空状态，初始值
    GRANTED, //锁被授予的状态
    VICTIM, //某个会话被选为了受害者
    TIMEOUT, //超时状态，申请加锁却发生超时
    KILLED }; //被killed状态，发起锁请求的会话被killed掉，所以不仅是发起加锁请求的事务应终止，事务的属主会话也被终止
...}
```

## 5. 元数据锁类对象

保存每一种 MDL\_key 对应的所有已经授予的 MDL 锁信息，因为 MDL\_key 标识了不同的数据库对象类，不同的数据库对象类上所施加的锁之间的兼容性因对象存在差别，所以定义了不同的策略来区分这些差别。请注意，MDL\_lock 不是一个具体的锁，而是一类锁的集合。GLOBAL 和 COMMIT 类的锁，是全局唯一的，除此之外的其他类型的锁，可以有多个。

```
/**
The lock context. Created internally for an acquired lock.
For a given name, there exists only one MDL_lock instance, and it exists only
when the lock has been granted.
Can be seen as an MDL subsystem's version of TABLE_SHARE. //能够被存储层的TABLE_
SHARE使用这个锁对象

This is an abstract class which lacks information about compatibility rules for lock types.
They should be specified in its descendants.
*/
class MDL_lock //当需要施加元数据锁的时候，生成一个MDL_lock锁对象
{...
    class Ticket_list{...} //把MDL_ticket封装为一个 List对象，用以存放多个MDL_ticket
    锁对象（MDL_ticket参见下一小节）

/** //对于锁的操作，又存在两种策略，这是根据被加锁的对象的特定区分的。每一种策略分别有各自的锁兼容规则
    Helper struct which defines how different types of locks are handled for a
    specific MDL_lock.
    In practice we use only two strategies:
    "scoped" lock strategy for locks in GLOBAL, COMMIT, TABLESPACE and SCHEMA
    namespaces //范围锁策略：范围带有空间的意味
    and "object" lock strategy for all other namespaces.
```





```

//对象锁策略：数据库内部的各种对象
*/
struct MDL_lock_strategy //锁的施加策略。如上所述，锁被分为两种类型，所以统一用策略数
据结构来管理和描述这两类锁的特性
{
    /**Compatibility (or rather "incompatibility") matrices for lock types.
    //新申请的锁向已经授予的锁进行锁的相容性判断
    Array of bitmaps which elements specify which granted locks
    are incompatible with the type of lock being requested.*/
    bitmap_tm_granted_incompatible[MDL_TYPE_END]; //已经被授予的锁的数组中保存有
    不兼容的、准备申请此对象的请求锁，位图数组结构

    //新申请的锁向已经正在等待的锁进行锁的相容性（优先级）判断。此数组的作用有两个：
    //一是通过m_waiting_incompatible[0],
    //二是防止产生锁饥饿现象，
    //所以增加了对低优先级锁的申请次数的计数，当计数到一定程度时，唤醒低优先级的尚没获得锁的会话。
    //可以关注MDL_lock::reschedule_waiters()函数，查看对等待的锁的处理方式；看其调用
    者查看唤醒条件。
    //另外看此函数中封锁粒度较强的锁释放而封锁粒度较弱的锁得以有机会被授予的时候，
    //m_hog_lock_count/m_piglet_lock_count有机会被重置
    //（注意强弱是相对的，取决于11.4.1节中第3小节中定义的enum_mdl_type中的锁的粒度值，
    据这些值比较大小）
    /** Arrays of bitmaps which elements specify which waiting locks
    are incompatible with the type of lock being requested.
    Basically, each array defines priorities between lock types.
    We need 4 separate arrays since in order to prevent starvation for
    some of lock request types, we use different priority matrices:
    0) in "normal" situation. //正常优先级
    1) in situation when the number of successively granted "piglet"
    requestsexceeds the max_write_lock_count limit. //小猪优先级
    2) in situation when the number of successively granted "hog"
    requestsexceeds the max_write_lock_count limit. //猪优先级
    3) in situation when both "piglet" and "hog" counters exceed limit.*/
    //小猪和猪之和
    //这个矩阵是某个锁请求与处于等待状态的锁的优先级比较表
    //第一个数组是正常情况，其他三个数组是为了解锁饥饿而设置的
    //如m_piglet_lock_count的值大于了max_write_lock_count，则m_waiting_
    incompatible[1][x]被置位
    //如m_hog_lock_count的值大于了max_write_lock_count，则m_waiting_
    incompatible[2][x]被置位
    //在等待的锁的数组中保存有不兼容的、准备申请此对象的请求锁，二维位图数组结构
    bitmap_tm_waiting_incompatible[4][MDL_TYPE_END]; //piglet，会申请SW锁；hog，
    会申请X、SNRW、SNW这三者其一

    /**Array of increments for "unobtrusive" types of lock requests for locks.
    @sa MDL_lock::get_unobtrusive_lock_increment().*/

```





```

// “unobtrusive” 类型相关的锁粒度包括：S、SH、SR 和SW，对应 “Fast path” 的访问方式，主要用于DML类操作
// “obtrusive” 类型相关的锁粒度包括：SU、SRO、SNW、SNRW、X，对应 “slow path” 的访问方式，主要用于非DML类操作
fast_path_state_tm_unobtrusive_lock_increment[MDL_TYPE_END];
//快速访问 “unobtrusive” 类型的锁

/**Indicates that locks of this type are affected by the max_write_lock_count limit.*/
bool m_is_affected_by_max_write_lock_count;

/**Pointer to a static method which determines if the type of lock requested requires notification of conflicting locks.
  NULL if there are no lock types requiring notification.*/
//当有冲突锁的时候，是否要被通知。“scoped lock” 不要求被通知，而 “object lock” 施加排它锁时才需要被通知
bool (*m_needs_notification)(const MDL_ticket *ticket);

/**Pointer to a static method which allows notification of owners of conflicting locks about the fact that a type of lock requiring notification was requested.*/
//对于 “object lock”，通知持有S、SH类锁的会话线程（可能与待定的X锁冲突，pending lock）
void (*m_notify_conflicting_locks)(MDL_context *ctx, MDL_lock *lock);
//发出通知的函数，含义类似上面

/**Pointer to a static method which converts information about locks granted using “fast” path from fast_path_state_t representation to bitmap of lock types.*/
//S、SR、SW粒度的锁可以被使用 “fast path” 方式快速处理
bitmap_t (*m_fast_path_granted_bitmap)(const MDL_lock &lock);

/**Pointer to a static method which determines if waiting for the lock should be aborted when connection is lost. NULL if locks of this type don't require such aborts.*/ //当连接断开的时候，锁是否应该被撤销。
//LOCKING_SERVICE和USER_LEVEL_LOCK加锁的情况可被撤销，如通过GET_LOCK()施加的锁。
bool (*m_needs_connection_check)(const MDL_lock *lock);
};

public:
/** The key of the object (data) being protected. */
MDL_key key; //元数据锁所属的类型（在MDL_key中把元数据这样的对象分为了几类，给每类定义一个key在enum_mdlnamespace枚举中）
...

/** List of granted tickets for this lock. */
Ticket_list m_granted; //对于本锁而言，在系统内部存在的已经被授予的所有锁list
/** Tickets for contexts waiting to acquire a lock. */

```





```

Ticket_list m_waiting; //对于本锁而言，在系统内部存在的正在等待被授予的所有锁list
//如上两个list，配合使用：
//当要获取一个锁（入通过acquire_lock()函数）不成功时，把新生成的一个MDL_ticket对象放入等待
//队列；获取成功，则放入m_granted中
//当一个处于等待状态的锁可以被授予的时候（can_grant_lock()判断是否可以被授予），就从m_
//waiting中remove掉，然后加入到m_granted中，
//这样的事情，当获取锁或释放锁时，或因ALTER TABLE等操作需要降级锁时，通过reschedule_
//waiters()函数进行
...
    /** Pointer to strategy object which defines how different types of lock
        requests should be handled for the namespace to which this lock belongs.
        @sa MDL_lock::m_scoped_lock_strategy and MDL_lock::m_object_lock_strategy. */
    const MDL_lock_strategy *m_strategy; //注意这是一个指针，执行哪个类型的策略就表示使用
    //被指向的策略对象之策略（指向下面两个策略对象之一）
...
    static const MDL_lock_strategy m_scoped_lock_strategy; //范围锁对应的策略
    static const MDL_lock_strategy m_object_lock_strategy; //对象锁对应的策略
};

```

## 6. 被授予的元数据锁对象凭证

要求被授予的 MDL 锁对象，即此锁可被持有（但不是一定已经持有），表明这是一种状态，一种可持有锁的状态或凭证。这样的锁的状态或持锁凭证，使用“MDL\_ticket”表示。

```

/**
A granted metadata lock.
@warning MDL_ticket members are private to the MDL subsystem.
@note Multiple shared locks on a same object are represented by a single ticket.
The same does not apply for other lock types.

@note There are two groups of MDL_ticket members: //“持锁”又被分为两种，注意其差别
- "Externally accessible". These members can be accessed from
//会话线程间共享，即可被多个会话线程访问
threads/contexts different than ticket owner in cases when
ticket participates in some list of granted or waiting tickets
for a lock. Therefore one should change these members before
including them to waiting/granted lists or while holding lock
protecting those lists.
- "Context private". Such members are private to thread/context //会话线程私有
owning this ticket. I.e. they should not be accessed from other
threads/contexts.

*/
class MDL_ticket : public MDL_wait_for_subgraph //可被授予的锁，继承了等待图，所以可
//以对其进行死锁检测
{...
    /**Pointers for participating in the list of lock requests for this context.

```



```

Context private. */
MDL_ticket *next_in_context; //把在同一个MDL_context上下文中的所有MDL_ticket用双向
链表串起来
MDL_ticket **prev_in_context; //此双向链表被MDL_context作为迭代器“Ticket_
list::Iterator Ticket_iterator”使用
/**Pointers for participating in the list of satisfied/pending requests for the
lock. Externally accessible.*/
MDL_ticket *next_in_lock; //把在同一个MDL_lock中的所有MDL_ticket用双向链表串起来
MDL_ticket **prev_in_lock; //此双向链表被MDL_lock定义为了“Ticket_list”使用

/**Status of lock request represented by the ticket as reflected in P_S. */
enum enum_psi_status { PENDING = 0, GRANTED, PRE_ACQUIRE_NOTIFY, POST_RELEASE_
NOTIFY }; //在psi⊖系统中表示锁的状态

/**Context of the owner of the metadata lock ticket. Externally accessible.*/
MDL_context *m_ctx; //拥有元数据锁的属主的上下文，此上下文中定义了一些重要方法，详情参
见本小节中的“7 元数据锁的属主上下文”

/**Pointer to the lock object for this lock ticket. Externally accessible.*/
MDL_lock *m_lock; //元数据锁对象，其中定义了两种策略，非常重要，详情参见本小节中的
“8元数据锁对象”
...}

```

MDL\_ticket 对象，是 MDL\_lock 对象的一种状态，代表 MDL\_lock 对象作为等待图中的一个点，只有这样的点加入等待图，才能引发其他的点到这个点之间进行连线，只有用线把点之间连接起来，才有可能构成环，这就是 MDL\_ticket 所起的作用（凑成等待图的子部分）。

MDL\_ticket 对象上拥有 MDL\_lock 对象，表明在一个锁对象 MDL\_lock 上有了新的状态，这个新的状态就是本锁对象处于“已被授予了锁”这种状态，即这个锁由申请状态变为了授予状态。这个时候，就需要对锁链进行死锁检测了，因为每新被授予的锁可能导致环形成。

## 7. 元数据锁的上下文空间

对哪些锁对象进行操作，如围绕元数据的锁施加和释放、死锁检测等相关操作，被定义在 MDL\_context 类中，这是对元数据锁对象的操作空间（锁的相关操作，一定是以一个会话为主体，为这个主体申请锁释放锁，以这个主体为依托进行本主体和其他主体之间锁争用进行死锁检测，死锁检测后被牺牲的一定是某个主体）。这个空间，在一个会话（MySQL 内部通常称为 connection）中，用于管理本会话内部的所有的 MDL\_lock。

```

/** Context of the owner of metadata locks. I.e. each serverconnection has such a context.*/
class MDL_context //元数据锁的环境，即元数据锁的上下文，不是一个具体的元数据锁对象，而是元
数据锁的周围环境，所以定义了很多相关操作如加锁、释放锁等
{

```

<sup>⊖</sup> psi，是performance schema interface。





```

public:
...
    bool try_acquire_lock(MDL_request *mdl_request); //尝试加元数据锁
    bool acquire_lock(MDL_request *mdl_request, ulong lock_wait_timeout);
    //加单个元数据锁
    bool acquire_locks(MDL_request_list *requests, ulong lock_wait_timeout);
    //一次性批量加多个元数据排它锁, 用于RENAME、DROP等操作
    bool upgrade_shared_lock(MDL_ticket *mdl_ticket, //升级共享锁为排它锁, 不存在并发竞争才可升级
                             enum_mdl_type new_type,
                             ulong lock_wait_timeout);
...
    void release_all_locks_for_name(MDL_ticket *ticket);
    //从一堆元数据锁中释放指定元数据锁
    void release_lock(MDL_ticket *ticket);
    //释放特定元数据锁
...
    void release_statement_locks();
    //释放语句级元数据锁
    void release_transactional_locks();
    //释放事务级元数据锁
    void rollback_to_savepoint(const MDL_savepoint &mdl_savepoint);
    //回滚到指定保存点
...
public:
    /** If our request for a lock is scheduled, or aborted by the deadlock
        detector, the result is recorded in this class. */
    MDL_wait m_wait;
private:
    /**
        Lists of all MDL tickets acquired by this connection.
        //本会话已经获得的所有的元数据锁

        Lists of MDL tickets:
        -----
        The entire set of locks acquired by a connection can be separated
        //按照锁被释放的情况把锁又划分为三类
        in three subsets according to their duration: locks released at the end of statement,
        at the end of transaction and locks are released explicitly.

        Statement and transactional locks are locks with automatic scope.
        //上述三种锁的前两种, 是数据库引擎自动管理的,
        They are accumulated in the course of a transaction, and released
        //事务或语句结束就释放
        either at the end of uppermost statement (for statement locks) or
        on COMMIT, ROLLBACK or ROLLBACK TO SAVEPOINT (for transactional
        locks). They must not be (and never are) released manually,
        i.e. with release_lock() call.

        Tickets with explicit duration are taken for locks that span
    
```





//上述三种锁的第三种，是用户主动管理的，  
multiple transactions or savepoints.  
//如用户可以调用RELEASE\_LOCK()函数释放锁  
These are: HANDLER SQL locks (HANDLER SQL is  
transaction-agnostic), LOCK TABLES locks (you can COMMIT/etc  
under LOCK TABLES, and the locked tables stay locked), user level  
locks (GET\_LOCK()/RELEASE\_LOCK() functions) and  
locks implementing "global read lock".

Statement/transactional locks are always prepended to the  
beginning of the appropriate list. In other words, they are  
**stored in reverse temporal order**. Thus, when we rollback to  
//注意数据结构，反向存储，所以front()方法的使用  
a savepoint, we start popping and releasing tickets from the  
front until we reach the last ticket acquired after the savepoint.

Locks with explicit duration are not stored in any  
particular order, and among each other can be split into  
four sets://上述三种锁的第三种，又细分为四类

- LOCK TABLES locks
- User-level locks
- HANDLER locks
- GLOBAL READ LOCK locks \*/

Ticket\_list m\_tickets[MDL\_DURATION\_END];

//本会话内部所有的已经授予的MDL锁，注意是数组列表，数组根据锁的生命周期分为三类  
MDL\_context\_owner \*m\_owner;

...

/\*\*Tell the deadlock detector what metadata lock or tabledefinition cache  
entry this session is waiting for.

In principle, this is redundant, as information can be found by inspecting  
waiting queues,

but we'd very much like it to be readily available to the wait-for graph iterator.\*/

MDL\_wait\_for\_subgraph \*m\_waiting\_for; //等待图对象，表明本会话在等什么元数据锁

...

public:

void find\_deadlock(); //从m\_waiting\_for出发找出死锁(m\_waiting\_for表明有锁等待)，  
即从死锁的环中找出受害者。

//A fragment of recursive traversal of the wait-for graph of MDL contexts in  
the server in search for deadlocks.

bool visit\_subgraph(MDL\_wait\_for\_graph\_visitor \*dvisitor);

/\*\* Inform the deadlock detector there is an edge in the wait-for graph. \*/

void will\_wait\_for(MDL\_wait\_for\_subgraph \*waiting\_for\_arg)

{...}

...

};



## 8. 元数据锁的集合

MDL\_map 是元数据锁对象的集合。整个数据库系统产生的锁，都会被注册在锁集合 MDL\_map 中，便于寻找和管理。

```
/**A collection of all MDL locks. A singleton, there is only one instance of the
map in the server.*/
class MDL_map    //锁集合，MDL_key到MDL_lock的一个映射。所有的元数据锁都在此中注册
{...
    inline MDL_lock *find(LF_PINS *pins, const MDL_key *key, bool *pinned);
    //查找指定key值的锁是否在map中存在
    inline MDL_lock *find_or_insert(LF_PINS *pins, const MDL_key *key, bool
        *pinned); //查找指定key值的锁，不存在则创建
    ...
};
```

从下一节的分析可以看出（下一节将从锁系统到一个会话上的锁空间以及在所空间内的单个的锁整个生命周期来介绍锁的生命周期），MDL\_map 所定义的全局变量 mdl\_locks 是全局的锁入口。

## 9. 锁等待图

MySQL Server 层定义了用于死锁检测时使用的等待图，这和记录级锁死锁检测所使用的等待图的原理是一样，但是实现方式不同。

```
/**An abstract class for inspection of a connectedsubgraph of the wait-for graph.*/
class MDL_wait_for_graph_visitor //visitor, 访问者，意味着是执行某些动作的主体，所以内
部定义有enter_node()等操作类型的方法
{...} //定义了一些抽象的方法，实则作为一个接口（面向对象的编程思维）使用
```

定义了等待图中的“边”对象，在边对象中定义了受害者选取策略（主要是被选为受害者的权重值）。

```
/**Abstract class representing an edge in the waiters graph to be traversed by
deadlock detection algorithm.*/
class MDL_wait_for_subgraph
{...
    //定义了两个节点（持锁节点和请求锁节点）之间存在的边的权重，边表示请求锁的节点发出的请求类型，
    不同的类型有不同的权重，便于从所有请求中找出受害者。这实际上是定义了元数据锁的受害者选取的规则之一。
    static const uint DEADLOCK_WEIGHT_DML= 0;    //DML类被选为受害者的可能性最大
    static const uint DEADLOCK_WEIGHT_ULL= 50;    //对应用户层锁（MDL_key中的USER_
    LEVEL_LOCK）
    static const uint DEADLOCK_WEIGHT_DDL= 100;    //DDL类被选为受害者的可能性最小
    ...} //定义了一些抽象的方法，实则作为一个接口（面向对象的编程思维）使用
```

之前介绍的 MDL\_ticket 和没有介绍过的 Wait\_for\_flush 类继承了等待图类 MDL\_wait\_for\_subgraph。

## 10. 其他对象

MySQL 定义了 MDL\_context\_visitor 类，意在表示对元数据锁上下文 (MDL\_context) 的检查和访问，但除了做单元测试被使用外，其他代码处没有使用，不再详细介绍。

MDL\_context\_owner 类，表示元数据锁上下文 (MDL\_context) 的属主，每一个会话的线程就是一个 MDL\_context 对象的属主。

MDL\_wait 类，表示锁的等待状态。

## 11. 相容性

MDL\_lock::MDL\_lock\_strategy 的 MDL\_lock::m\_scoped\_lock\_strategy 定义的相容性如表 11-11 和 11-12 所示。

表 11-11 scoped lock 的已经授予的锁和新的申请锁相容性矩阵表

		GrantedMode, 已经授予的锁 /scoped lock			
		IS(*)	IX	S	X
Requested Mode 正申请的锁	IS	Y	Y	Y	
	IX	Y	Y		
	S	Y		Y	
	X	Y			

说明：

- ❑ 本表表明在同一个数据项上不同事务之间的加锁操作的并发情况。
- ❑ 注意 IS、IX 等均为缩写，其含义参考“3. 元数据锁的粒度”一节。
- ❑ cell 值“Y”表示可以授予锁；否则不可以授予。

表 11-12 scoped lock 的处于等待状态的锁和新的申请锁相容性矩阵表

		Pending requests for lock/, 处于等待状态的 scoped 锁			
		IS(*)	IX	S	X
Requested Mode 正申请的锁	IS	Y	Y	Y	Y
	IX	Y	Y		
	S	Y	Y	Y	
	X	Y	Y	Y	Y

说明：

- ❑ 本表表明在同一个数据项上不同事务之间的加锁操作的并发情况。
- ❑ cell 值“Y”表示可以授予锁；否则不可以授予。

MDL\_lock::MDL\_lock\_strategy 的 MDL\_lock::m\_object\_lock\_strategy 定义的相容性如表 11-13 和 11-14 所示。



表 11-13 object lock 的已经授予的锁和新的申请锁相容性矩阵表

		Granted Mode, 已经授予的 object 锁									
		S	SH	SR	SW	SWLP	SU	SRO	SNW	SNRW	X
Requested Mode 正申请的锁	S	Y	Y	Y	Y	Y	Y	Y	Y	Y	
	SH	Y	Y	Y	Y	Y	Y	Y	Y	Y	
	SR	Y	Y	Y	Y	Y	Y	Y	Y		
	SW	Y	Y	Y	Y	Y	Y				
	SWLP	Y	Y	Y	Y	Y	Y				
	SU	Y	Y	Y	Y		Y				
	SRO	Y	Y	Y		Y	Y	Y	Y		
	SNW	Y	Y	Y				Y			
	SNRW	Y	Y								
	X										

说明：

- ❑ 本表表明在同一个数据项上不同事务之间的加锁操作的并发情况。
- ❑ cell 值“Y”表示可以授予锁；否则不可以授予。

表 11-14 object lock 的处于等待状态的锁和新的申请锁相容性矩阵表

		Pending requests for lock, 处于等待状态的 object 锁									
		S	SH	SR	SW	SWLP	SU	SRO	SNW	SNRW	X
Requested Mode 正申请的锁	S	Y	Y	Y	Y	Y	Y	Y	Y	Y	
	SH	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
	SR	Y	Y	Y	Y	Y	Y	Y	Y		
	SW	Y	Y	Y	Y	Y	Y	Y			
	SWLP	Y	Y	Y	Y	Y	Y				
	SU	Y	Y	Y	Y	Y	Y	Y	Y	Y	
	SRO	Y	Y	Y	Y	Y	Y	Y	Y		
	SNW	Y	Y	Y	Y	Y	Y	Y	Y	Y	
	SNRW	Y	Y	Y	Y	Y	Y	Y	Y	Y	
	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

说明：

- ❑ 本表表明在同一个数据项上不同事务之间的加锁操作的并发情况。
- ❑ cell 值“Y”表示可以授予锁；否则不可以授予。

11.4.2 元数据锁的管理与使用

对于元数据锁的管理，相对记录锁的管理，更为复杂，这是因为：

- ❑ 从广度看，涉及的内容更多，包括各种对象、各种对象上不同类型的锁的管理，而记录锁的管理涉及的只是数据一种对象，这一点从上一节讲述的众多数据结构就可

以看出。

□从深度看上，同样包括锁的申请、施加、释放和死锁检测等过程，但因各种对象混杂，使得锁的申请、施加、释放和死锁检测等过程需要因对象而异，这样增加了理解的困难。

举个例子，MDL\_lock类就因为对象不同而分出两种管理策略（MDL\_scoped\_lock\_strategy和MDL\_object\_lock\_strategy）。为了讲述地更明白，将从系统初始化开始、按照元数据锁的生命周期过程，从前到后逐步展开，每一个阶段将包括相关对象和辅助操作的重点说明。所以请注意如下讲述的顺序就是一个有次序有关联关系的“逻辑”顺序。

## 1. 锁系统的初始化

MySQL在启动的过程中，如果init\_server\_components()->mdl\_init()->mdl\_locks.init()完成元数据锁系统的初始化工作。其中，mdl\_locks是一个全局的MDL\_map对象。

```
void MDL_map::init()/** Initialize the container for all MDL locks. */
//在mysqld启动时初始化MDL模块，作为全局信息
{
    MDL_key global_lock_key(MDL_key::GLOBAL, "", ""); //定义全局锁
    MDL_key commit_lock_key(MDL_key::COMMIT, "", ""); //定义全局提交锁

    m_global_lock= MDL_lock::create(&global_lock_key);
    //创建全局锁，在mdl_locks全局MDL_map对象中存在
    m_commit_lock= MDL_lock::create(&commit_lock_key);
    //创建全局提交锁，在mdl_locks全局MDL_map对象中存在

    m_unused_lock_objects= 0;
    //m_locks，无锁HASH(lock free⊖)，在mdl_locks全局MDL_map对象中存在，因无锁结构能
    大幅提高并发度，效率高
    lf_hash_init2(&m_locks, sizeof(MDL_lock), LF_HASH_UNIQUE,
        0, 0, mdl_locks_key, &my_charset_bin, &murmur3_adapter,
        &mdl_lock_cons, &mdl_lock_dtor, &mdl_lock_reinit);
}
```

所以，数据库引擎初始化后，全系统存在的是一个MDL\_map类型的mdl\_locks对象，后续的工作将围绕此对象上的m\_locks、m\_global\_lock和m\_commit\_lock展开。

其中m\_locks最为重要，后续操作中需要申请的MDL锁，都要从m\_locks上通过MDL\_map::find\_or\_insert()函数查找是否存在，不存在所申请的元数据锁，才创建新锁。

m\_locks作为一个Hash表，其中的每一个元素都是MDL\_lock对象，即每一个元素都是一个“元数据锁”。

全局的mdl\_locks对象在数据库引擎结束的时候通过调用clean\_up(bool print\_message)->mdl\_destroy()->mdl\_locks.destroy()完成对象的释放。

⊖ 无锁技术，可参考论文《Split-Ordered Lists: Lock-Free Extensible Hash Tables》



## 2. 元数据锁的上下文的初始化

上节说明 `mdl_locks` 是一个全局的 `MDL_map` 对象，全局的含义是指整个数据库引擎，在一个数据库引擎内部，元数据锁的操作范围，通常是在一个会话（SESSION，MySQL 称为一个 connection 即连接）内部进行的，所以当会话建立的时候，一个 `MDL_context` 对象就会生成，这个 `MDL_context` 对象将伴随着会话的整个生命周期。

```
class MDL_context
{...
void init(MDL_context_owner *arg) { m_owner= arg; } //MDL_context对象初始化
...}
```

一个会话建立的时候，一个 `MDL_context` 对象就会生成。

```
class THD :public MDL_context_owner, //THD线程，即一个会话，一个连接，继承了MDL_
context对象的属主MDL_context_owner
    public Query_arena,
    public Open_tables_state
{...
public:
    MDL_context mdl_context; //会话定义了自己的MDL_context对象，在THD被初始化的时候
    mdl_context被初始化
    ...
}
```

此后，如果在这个会话上执行 SQL 语句，就根据 SQL 语句的加锁语义，对锁进行申请，数据库引擎就会通过事务锁的封锁机制，确定加锁释放能够成功。

## 3. 锁的申请

如果在一个对象上发出加锁请求（SQL 语句发出），需要构造加锁需要的“元数据锁的锁请求”对象 `MDL_request`，然后交给“元数据锁的上下文” `MDL_context` 的获取锁函数 `acquire_locks()` 来加锁。

例如创建事件是需要在指定对象上发出加锁请求，方式如下：

```
bool lock_object_name(THD *thd, MDL_key::enum_mdl_namespace mdl_type, const char
*db, const char *name)
{
    MDL_request_list mdl_requests;
    MDL_request global_request;
    MDL_request schema_request;
    MDL_request mdl_request;
    ...
    //构造加锁请求。加锁请求，体现了SQL语句的封锁意愿，以及并发情况下，是否存在其他加锁意愿时需要构造不同的加锁请求
    MDL_REQUEST_INIT(&global_request,
```

```

//为global_request锁请求在GLOBAL空间内申请语句级的意向排它锁
    MDL_key::GLOBAL, "", "", MDL_INTENTION_EXCLUSIVE, MDL_STATEMENT);
MDL_REQUEST_INIT(&schema_request, //为schema_request锁请求在SCHEMA空间内申请事
务级的意向排它锁
    MDL_key::SCHEMA, db, "", MDL_INTENTION_EXCLUSIVE, MDL_TRANSACTION);
MDL_REQUEST_INIT(&mdl_request, //为mdl_request锁请求在参数mdl_type空间内申请
事务级的排它锁
    mdl_type, db, name, MDL_EXCLUSIVE, MDL_TRANSACTION);

mdl_requests.push_front(&mdl_request);
mdl_requests.push_front(&schema_request);
mdl_requests.push_front(&global_request); //把新的加锁请求放到链表的前端

if (thd->mdl_context.acquire_locks(&mdl_requests,
//在会话thd对象的元数据锁上下文 (MDL_context) 空间内获取准备施加的锁
    thd->variables.lock_wait_timeout))
    //设置超时等待

return TRUE;
...
}

```

与 lock\_object\_name() 函数相似的能够发出加锁申请的函数还有 lock\_schema\_name()、lock\_tablespace\_name()、acquire\_locking\_service\_locks()、lock\_fk\_dependent\_tables()、lock\_table\_names()、lock\_db\_routines()、mysql\_alter\_table(), 都是使用宏 MDL\_REQUEST\_INIT 来构造加锁的请求, 然后交由 acquire\_locks() 函数来完成加锁。

其他加锁请求, 则通过调用 acquire\_lock() 函数来完成加锁。

如下以 “ALTER TABLE t1 ROW\_FORMAT = COMPRESSED;” 执行栈为例, 来看表级锁 (表级锁是元数据锁之一) 的施加情况:

首先, “ALTER TABLE...” 语句调用 mysql\_alter\_table() 函数执行本命令, 在执行的过程中, 因为语义是在表级上做操作, 所以需要先通过 open\_tables() 函数打开表。而打开表的过程, 就需要判断是有资格否能够在表上执行操作, 这就需要判断权限和事务的元数据锁是否相容, 所以如下首先要通过 acquire\_lock() 函数获取 GLOBAL 类型的语句级表意向排它锁。

```

MDL_context::try_acquire_lock_impl(MDL_request * mdl_request, MDL_ticket * * out_ticket)
//申请MDL_INTENTION_EXCLUSIVE
MDL_context::acquire_lock(MDL_request * mdl_request, unsigned long lock_wait_timeout)
open_table(THD * thd, TABLE_LIST * table_list, Open_table_context * ot_ctx)
open_and_process_table(THD * thd, LEX * lex, TABLE_LIST * tables, unsigned int
* counter, unsigned int flags, Prelocking_strategy * prelocking_strategy, bool
has_prelocking_list, Open_table_context * ot_ctx)
open_tables(THD * thd, TABLE_LIST * * start, unsigned int * counter, unsigned int
flags, Prelocking_strategy * prelocking_strategy) mysql_alter_table(THD * thd,

```



```

const char * new_db, const char * new_name, st_ha_create_information * create_
info, TABLE_LIST * table_list, Alter_info * alter_info)
Sql_cmd_alter_table::execute(THD * thd)
mysql_execute_command(THD * thd, bool first_level)
mysql_parse(THD * thd, Parser_state * parser_state)
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command command)
do_command(THD * thd)

```

其次，调用 `open_table_get_mdlock()` 函数、根据不同的打开表需求（需求来自于打开的表对象上的封锁语义，即 `table_list->lock_type`，其值为 11.5.1 节所述的 `thr_lock_type` 枚举所定义，所以我们说加锁的需求来自于 SQL 语句的语义）施加不同的元数据锁。

```

MDL_context::acquire_lock(MDL_request * mdl_request, unsigned long lock_wait_timeout) //申请
open_table_get_mdlock(THD * thd, Open_table_context * ot_ctx, TABLE_LIST *
table_list, unsigned int flags, MDL_ticket * * mdl_ticket) //属于MySQL Server层
open_table(THD * thd, TABLE_LIST * table_list, Open_table_context * ot_ctx)
open_and_process_table(THD * thd, LEX * lex, TABLE_LIST * tables, unsigned int
* counter, unsigned int flags, Prelocking_strategy * prelocking_strategy, bool
has_prelocking_list, Open_table_context * ot_ctx) //与上一个栈信息在此处位于同一行
open_tables(THD * thd, TABLE_LIST * * start, unsigned int * counter, unsigned int
flags, Prelocking_strategy * prelocking_strategy) mysql_alter_table(THD * thd,
const char * new_db, const char * new_name, st_ha_create_information * create_
info, TABLE_LIST * table_list, Alter_info * alter_info)
Sql_cmd_alter_table::execute(THD * thd)
mysql_execute_command(THD * thd, bool first_level)
mysql_parse(THD * thd, Parser_state * parser_state)
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command command)
do_command(THD * thd)

```

最后，同样会如 11.5.3 节中的第 3 小节描述进入到 InnoDB 的 `store_lock()` 和 `external_lock()` 函数中传递来自 MySQL Server 层的锁的类型等参数值。这表明 InnoDB 的锁信息源自 MySQL Server 层。

```

ha_innobase::external_lock(THD * thd, int lock_type)
handler::ha_external_lock(THD * thd, int lock_type)
lock_external(THD * thd, TABLE * * tables, unsigned int count)
mysql_lock_tables(THD * thd, TABLE * * tables, unsigned int count, unsigned int flags)
lock_tables(THD * thd, TABLE_LIST * tables, unsigned int count, unsigned int flags)
mysql_inplace_alter_table(THD * thd, TABLE_LIST * table_list, TABLE * table,
TABLE * altered_table, Alter_inplace_info * ha_alter_info, enum_alter_inplace_
result inplace_supported, MDL_request * target_mdlock_request, Alter_table_ctx * alter_ctx)
mysql_alter_table(THD * thd, const char * new_db, const char * new_name, st_ha_
create_information * create_info, TABLE_LIST * table_list, Alter_info * alter_info)
Sql_cmd_alter_table::execute(THD * thd) //与上一个栈信息在此处位于同一行
mysql_execute_command(THD * thd, bool first_level)
mysql_parse(THD * thd, Parser_state * parser_state)

```

```
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command command)
do_command(THD * thd)
```

#### 4. 锁的施加

每当要申请一个锁的时候，申请者必然是一个会话（此会话上正在执行的 SQL 语句发出的加锁申请），需要在会话所属的锁上下文空间中进行锁的申请操作。因此要调用 MDL\_context::acquire\_lock() 方法。

```
/** Acquire one lock with waiting for conflicting locks to go away if needed. */
bool
MDL_context::acquire_lock(MDL_request *mdl_request, ulong lock_wait_timeout)
//获取mdl_request锁请求
{...
//首先，第一步：找与锁请求可相容的已经存在的MDL_ticket锁⊖，如果存在则直接使用；找不到则创建
新的MDL_ticket对象。
//另外，在找MDL_ticket对象的时候，先看其所属的大类型MDL_lock是否存在（依据MDL_key），不存在
则先在MDL_map中创建
//MDL_lock，然后把MDL_ticket对象与MDL_lock类互相注册（互相用指针指向）
    if (try_acquire_lock_impl(mdl_request, &ticket)) //“在m_tickets 范围内”⊖找
        mdl_request锁请求相近的锁；找不到则创建新的
            return TRUE;//找到一个已经存在的锁和申请的锁匹配
//相近是指MDL_key相同，且申请的锁相容（可升级或与存在的锁相等）；但锁的生命周期可不同（由
枚举enum_mdL_duration定义）

    if (mdl_request->ticket)//try_acquire_lock_impl()中成功创建了一个锁对象，此锁则可
        被授予，所以是MDL_ticket对象
    {
        /*We have managed to acquire lock without waiting.
        MDL_lock, MDL_context and MDL_request were updated accordingly, so we
        can simply return success.*/
        return FALSE;
    }

//其次，第二步：从这之后的语义是：锁尚不能获取，处于等待状态
    /*Our attempt to acquire lock without waiting has failed.
    As a result of this attempt we got MDL_ticket with m_lockmember pointing
```

⊖ 例如，执行如下SQL：

```
S1 BEGIN;
S2 INSERT INTO t1 VALUES (1);
S3 INSERT INTO t1 VALUES (2);
```

S2首先会创建一个新的元数据锁，S3想要获取的元数据锁和S2申请的元数据锁是一致的且这是一个事务内部的加锁操作，所以完全可以复用同一个元数据锁，这样就会在try\_acquire\_lock\_impl()中通过find\_ticket()来获取一个已经存在的锁复用。

⊖ “在m\_tickets 范围内”是指优先找有相同生命周期的同类锁，找不到在遍历m\_tickets中的其他行（其中的锁有着不同的生命周期）



```

    to the corresponding MDL_lock object
    which has MDL_lock::m_rwlock write-locked.*/
lock= ticket->m_lock;//把MDL_ticket对象与MDL_lock类互相注册（互相用指针指向）后的引用
lock->m_waiting.add_ticket(ticket);//互相注册（互相用指针指向，此处是把ticket作为等待对象加入等待队列的尾部）⊖

```

//第三步：进行死锁检测

```

/*Once we added a pending ticket to the waiting queue,we must ensure that our
wait slot is empty, so that our lock
request can be scheduled. Do that in thecritical section formed by the
acquired write lock on MDL_lock.*/
m_wait.reset_status();

```

```

if (lock->needs_notification(ticket)) //object类型，非scoped类型；根据锁策略确定
（参加11.4.1的第5小节）

```

```

    lock->notify_conflicting_locks(this);

```

...

```

will_wait_for(ticket); //ticket处于等待状态，通知死锁检测器等待图是谁

```

...

```

find_deadlock(); //找死锁

```

//注意如下对于**timed\_wait()**的调用。不能立即获取的锁就需要进行超时等待判断

//即让申请的锁等待一段时间，看释放能够在指定的时间段内，因其他会话释放了锁而得到这样的锁

```

if (lock->needs_notification(ticket) || lock->needs_connection_check())

```

//object类型，非scoped类型

```

{...

```

```

    set_timespec(&abs_shortwait, 1); //要进行超时检测

```

```

    wait_status= MDL_wait::EMPTY; //初始状态

```

```

    while (cmp_timespec(&abs_shortwait, &abs_timeout) <= 0)

```

```

    {

```

```

        /* abs_timeout is far away. Wait a short while and notify locks. */

```

```

        wait_status= m_wait.timed_wait(m_owner, &abs_shortwait, FALSE,mdl_
request->key.get_wait_state_name());

```

```

        if (wait_status != MDL_wait::EMPTY)

```

```

            break;

```

...

```

    }

```

```

    if (wait_status == MDL_wait::EMPTY)

```

```

        wait_status= m_wait.timed_wait(m_owner, &abs_timeout, TRUE,mdl_
request->key.get_wait_state_name());

```

```

    }

```

```

else //scoped类型，非object类型

```

```

{

```

<sup>⊖</sup> 意味着将来使用m\_waiting时，是从其头部取对象，满足先进先出的要求，即队列。

```

        wait_status= m_wait.timed_wait(m_owner, &abs_timeout, TRUE,mdl_request->
        key.get_wait_state_name());
    }

    done_waiting_for(); //与will_wait_for(ticket)对应,表明目前不需要进行死锁检测(因为
    刚刚已完成一次死锁检测)
...
//第四步:死锁检测完成,进行状态检查
//在这之前,ticket因处于等待,可能被其他会话在释放了锁之后通过reschedule_waiters(),而被授
予了锁(并发的情况)
    if (wait_status != MDL_wait::GRANTED) //对锁的状态开始进行判断
    {
        lock->remove_ticket(this, m_pins, &MDL_lock::m_waiting, ticket);
        //锁不能被授予,则去掉之前注册到MDL_lock中的ticket对象
...

        MDL_ticket::destroy(ticket);
        //锁不能被授予,则去掉之前注册到MDL_lock中的ticket对象之后销毁它
        switch (wait_status) //锁不能被授予,对不能被授予的触发因素细化,通知用户原因
        {
            case MDL_wait::VICTIM: //锁不能被授予原因之一:死锁发生,找到了受害者
                my_error(ER_LOCK_DEADLOCK, MYF(0));
                break;
            case MDL_wait::TIMEOUT: //锁不能被授予原因之二:超时发生
                my_error(ER_LOCK_WAIT_TIMEOUT, MYF(0));
                break;
            case MDL_wait::KILLED: //锁不能被授予原因之三:申请锁对应的会话线程(用户连接)被killed了
                if (get_owner()->is_killed() == ER_QUERY_TIMEOUT)
                    my_error(ER_QUERY_TIMEOUT, MYF(0));
                else
                    my_error(ER_QUERY_INTERRUPTED, MYF(0));
                break;
            default: //锁不能被授予原因之:不可能有其他原因,如果能到这里,则数
                据库引擎存在bug

                DEBUG_ASSERT(0);
                break;
        }
        return TRUE;
    }
}

//第五步:不存在死锁,且没有发生超时,则锁被授予
/*We have been granted our request. //超时等待判断之后,不能授予锁的情况上面已加处
理。只剩下锁能被授予的情况
    State of MDL_lock object is already being appropriately updated by
    aconcurrent thread (@sa MDL_lock:reschedule_waiters()).
    So all we need to do is to update MDL_context and MDL_request objects. */
DEBUG_ASSERT(wait_status == MDL_wait::GRANTED);

```



```

m_tickets[mdl_request->duration].push_front(ticket);
//锁能被授予则注册到锁相关的MDL_context上下文空间里
mdl_request->ticket= ticket; //锁能被授予也注册到锁请求对象上
...
return FALSE;
}

```

如果想要获取的 ticket 对象所依赖的 MDL\_lock 对象不存在，则要先在引擎内部建立 MDL\_lock 对象，通过如下过程完成的。

```

#0 inline_mysql_prlock_init (key=19, that=0x4fble88)
   at /home/dbsrc/mysql-5.7.17/include/mysql/psi/mysql_thread.h:840
#1 0x0000000014bcbcb in MDL_lock::MDL_lock (this=0x4fbl00)
   at /home/dbsrc/mysql-5.7.17/sql/mdl.cc:802
#2 0x0000000014b74a2 in mdl_lock_cons (arg=0x4fblce0 "?\376\004")
//4调用MDL_lock的构造方法，新建一个MDL_lock锁对象
   at /home/dbsrc/mysql-5.7.17/sql/mdl.cc:1121
#3 0x000000001905e9b in lf_alloc_new (pins=0x4e98600)
   at /home/dbsrc/mysql-5.7.17/mysys/lf_alloc-pin.c:441
#4 0x0000000019071cb in lf_hash_insert (hash=0x2de2800 <mdl_locks>,
//3在MDL_map里的无锁Hash内找（找不到，则创建一个新的）
   pins=0x4e98600, data=0x4fded10) at /home/dbsrc/mysql-5.7.17/mysys/lf_hash.c:488
#5 0x0000000014bcf8f in MDL_map::find_or_insert (this=0x2de2800 <mdl_locks>,
//2 在MDL_map集合内找（找不到，则创建一个新的）
   pins=0x4e98600, mdl_key=0x4fded10, pinned=0x7fc2e85e8527) at /home/dbsrc/
mysql-5.7.17/sql/mdl.cc:1266
#6 0x0000000014b917b in MDL_context::try_acquire_lock_impl (this=0x4f4f968,
//1准备获取ticket对象
   mdl_request=0x4fdecf0, out_ticket=0x7fc2e85e85a8) at /home/dbsrc/
mysql-5.7.17/sql/mdl.cc:3083
#7 0x0000000014b9ba0 in MDL_context::acquire_lock (this=0x4f4f968,
   mdl_request=0x4fdecf0, lock_wait_timeout=31536000) at /home/dbsrc/
mysql-5.7.17/sql/mdl.cc:3564
#8 0x0000000014ba2e9 in MDL_context::acquire_locks (this=0x4f4f968,
   mdl_requests=0x7fc2e85e87c0, lock_wait_timeout=31536000) at /home/dbsrc/
mysql-5.7.17/sql/mdl.cc:3814

```

锁的获得，还需要经过一个重要的方法进行条件判断，即在数据库封锁理论中常说的锁的相容性判断。但是 MySQL 不仅要进行锁的相容性判断，还多了两个工业级的实现，一个是与等待的锁的优先级判断，一个是被称为“fast path”的相容判断（参见 11.8.2 节）。代码如下：

```

/**Check if request for the metadata lock can be satisfied given its current state.
  @param type_arg The requested lock type.
  @param requestor_ctx The MDL context of the requestor.
  @retval TRUE Lock request can be satisfied //锁可以被授予

```

```

@retval FALSE      There is some conflicting lock. //存在冲突，锁不可以被授予

@note In cases then current context already has "stronger" type of lock on the object
      it will be automatically granted thanks to usage of the MDL_
      context::find_ticket() method.

*/
bool //确认：加锁会话即加锁请求者requestor_ctx申请的type_arg类型锁是否可以被授予，这个判
断是在对象/锁主体上进行判断的
MDL_lock::can_grant_lock(enum_mdll_type type_arg, const MDL_context *requestor_ctx) const
{
    bool can_grant = FALSE;
    bitmap_t waiting_incompat_map = incompatible_waiting_types_bitmap()[type_arg];
    bitmap_t granted_incompat_map = incompatible_granted_types_bitmap()[type_arg];

    /*New lock request can be satisfied iff: //可被授予锁的两个条件
    - There are no incompatible types of satisfied requests in other contexts
    //其他会话中不存在不相容的锁
    - There are no waiting requests which have higher priority than this
    request.*/ //等待队列中没有比本请求优先级更高的
    if (!(m_waiting.bitmap() & waiting_incompat_map)) //等待的锁中没有比新申请的锁有
    更高的优先级，则可能获得批准
    { //上面这个条件判断的含义是：本锁的等待者已经比新申请的锁有更高优先级，则新申请者就没有资格
    “排队”等待了
        if (!(fast_path_granted_bitmap() & granted_incompat_map))
        {
            if (!(m_granted.bitmap() & granted_incompat_map))
            //已经授予的锁中没有比新申请的锁存在不相容的，则可获得批准
            can_grant = TRUE;
            else //以上均不满足，意味着存在不相容的问题
            {
                Ticket_iterator it(m_granted);
                MDL_ticket *ticket;

                /* There is an incompatible lock. Check that it belongs to
                some other context.
                If we are trying to acquire "unobtrusive" type of lock then
                the conflicting lock must be from "obtrusive"
                set, therefore it should have been acquired using "slow path"
                and should be present in m_granted list.

                If we are trying to acquire "obtrusive" type of lock then it
                can be either another "obtrusive" lock
                or "unobtrusive" type of lock acquired on "slow path" (can't
                be "unobtrusive" lock on fast path
                because of surrounding if-statement). In either case it
                should be present in m_granted list.*/

```



```

        while ((ticket= it++))
        {
            if (ticket->get_ctx() != requestor_ctx &&
                ticket->is_incompatible_when_granted(type_arg))
                break; //存在不相容的问题：不相容来自其他的会话，则本次锁申请只能等待
        }
        if (ticket == NULL) /* Incompatible locks are our own. */
        //上面的while循环遍历了m_granted中所有的ticket，没有在其他MDL_context中找到不相容的，则表明不相容的发生在同一个MDL_context中
            can_grant= TRUE; //是本次会话造成的不相容（本事务内）则不用等待，可授予
        }
    }
    else
    {
        /* Our lock request conflicts with one of granted "fast path" locks:
           This means that we are trying to acquire "obtrusive" lock and:
           a) Either we are called from MDL_context::try_acquire_lock_impl()
           and then all "fast path" locks
               belonging to this context were materialized (as we do for
           "obtrusive" locks).
           b) Or we are called from MDL_lock::reschedule_waiters() then this
           context is waiting for this request
               and all its "fastpath" locks were materialized before the wait.
           The above means that conflicting granted "fast path" lock cannot
           belong to us and our request cannot be satisfied.
           */
    }
}
return can_grant;
}

```

## 5. 锁的释放

当事务结束（提交或回滚），或语句级锁对应的 SQL 语句操作执行完成，或加锁失败等事件发生时，锁需要释放，这时会调用到“MDL\_context::release\_lock(enum\_mdL\_duration duration, MDL\_ticket \*ticket)”方法，在锁上下文空间内来释放锁。

```

void MDL_context::release_lock(enum_mdL_duration duration, MDL_ticket *ticket)
{
    ...
    if (ticket->m_is_fast_path) { ... //如果是快速的访问方式（不针对全局的MDL_key值为GLOBAL和COMMIT）
        /* Don't count singleton MDL_lock objects as unused. */
        if (last_use && ! is_singleton) //如果是快速的访问方式，不释放，只标记为不再被使用的状态，
            mdl_locks.lock_object_unused(this, m_pins); //这样其他的加锁请求可以继续使用
    }
    else //把已经授予的MDL_ticket从MDL_lock中去掉，注意MDL_ticket是要被释放的锁对象，对应着本方法的参数

```

```

{
    /*Lock request represented by ticket was acquired using "slow path"
    or ticket was materialized later. We need to use "slow path" release.*/
    //lock对象的remove_ticket()方法会调用reschedule_waiters()以让其他会话有机会获得锁
    //自己(lock对象)不再需要占用锁资源了,就要及时通知别人获取,主动帮助别人的方式更好哦
    lock->remove_ticket(this, m_pins, &MDL_lock::m_granted, ticket);
    //lock对象是注册在ticket对象上的MDL_lock
}

m_tickets[duration].remove(ticket); //还需要从锁上下文空间MDL_context中把注册过的ticket去掉
...
MDL_ticket::destroy(ticket); //当注册去掉后,曾经被授予的锁对象可以被销毁了
...
}

```

例如,当事务完成的时候,元数据锁的全局类(MDL\_key是GLOBAL和COMMIT)通过lock\_object\_unused()函数完成释放,而此函数主要被release\_lock()函数在事务提交/回滚的时候(ha\_commit\_trans()函数)调用,如图11-5所示。注意对于全局类的元数据锁,有一个蓄水水位的管理方式,可参见mdl\_locks\_unused\_locks\_low\_water变量和lock\_object\_unused()函数。

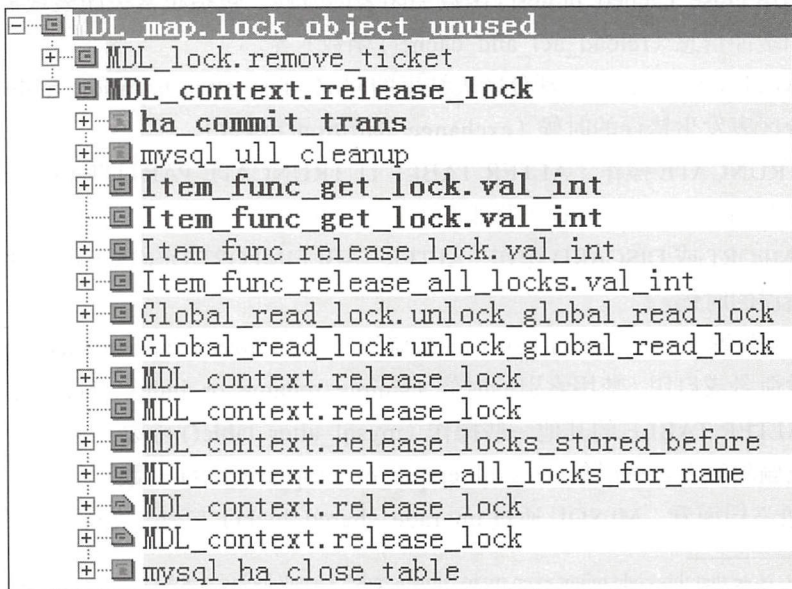


图 11-5 lock\_object\_unused() 函数调用关系图

## 6. 锁的降级和升级

锁的自动升降级操作是有效提升并发度的方法之一。MySQL 允许对锁进行自动降级操



作，但没有允许对锁进行自动升级操作（代码注释里说可升级<sup>①</sup>，指的是根据 SQL 操作的业务需要而利用锁的相容性性进行的 SQL 语义层面的升级，而不是为了提高并发度或减少因锁的使用而带来更多的系统 CPU 和内存等资源消耗而自动进行的锁的升级<sup>②</sup>，示例如图 11-6 所示）。

```
//Downgrade an EXCLUSIVE or SHARED_NO_WRITE lock to shared metadata lock.
void MDL_ticket::downgrade_lock(enum_mdl_type new_type)
{...
    /* Only allow downgrade from EXCLUSIVE and SHARED_NO_WRITE. */
    DEBUG_ASSERT(m_type == MDL_EXCLUSIVE || m_type == MDL_SHARED_NO_WRITE);
    //只允许对这两类锁进行降级，m_type表示旧的锁类型

    /* Below we assume that we always downgrade "obtrusive" locks. */
    DEBUG_ASSERT(m_lock->is_obtrusive_lock(m_type)); //只对“obtrusive”类型的锁降级
    ...

    m_lock->reschedule_waiters(); //锁被降级的时候，调用reschedule_waiters()可以给别的
    会话以获取锁的机会，能提高并发度
    ...
}
```

寻找锁可降级的机会，是通过锁降级提升并发度的关键。MySQL 允许在如下情况下，对封锁粒度大的排它锁或写锁进行降级：

- ❑ 通过调用 `close_cached_tables()` 函数关闭表的时候，包括正常关闭表或需要临时加载权限和缓冲信息（`reload_acl_and_cache()` 函数）等。
- ❑ 执行 `ALTER TABLE` 时处理分区表结束的间隙（`fast_alter_partition_table()` 函数）。
- ❑ 处理分区表发生错误的时候（`exchange_partition()` 函数）。
- ❑ 执行 `TRUNCATE` 操作（`ALTER TABLE t1 TRUNCATE PARTITION...`）时操作过程被写入 `binary log` 之后，锁可降级。
- ❑ 执行 `IMPORT` 或 `DISCARD` 操作（`ALTER TABLE IMPORT/DISCARD TABLE SPACE...`）发生错误的时候。
- ❑ 存储引擎告诉 MySQL server 层可以对锁降级的时候（`mysql_inplace_alter_table()` 函数）。
- ❑ 对表重命名或启用 / 禁用索引的时候（`simple_rename_or_index_change()` 函数）。
- ❑ 执行 `ALTER TABLE` 的其他一些操作（`mysql_alter_table()` 函数）。
- ❑ 创建或删除触发器的时候（`mysql_create_or_drop_trigger()` 函数）。

从 SQL 语义层面看，MySQL 通过 `upgrade_shared_lock()` 方法，支持对锁的升级。把锁

① 注释为：Note that this code might even try to "downgrade" a weak lock(e.g. SW) to a stronger one (e.g. SNRW).

② 锁的自动升级，是数据库引擎为了提高数据库引擎整体的运行效率而进行的一种自我调优的过程。例如，一个数据页面上加很多行级锁给很多不同的会话，很多锁导致死锁检测时间花费很多，则存在一种可能是：每个行级锁串行执行可能效率更高，那么，就可以为此种情况下的行级锁进行“自动”升级，升级到页面级锁，使得并发度降低但 CPU 和内存的消耗可能减少而让数据库引擎更为高效。这与 VoltDB 等数据库去掉并发的锁机制而改用序列化抑制并发的思路有相似之处。可参考论文《OLTP Through the Looking Glass, and What We Found There》。

升级到排它锁粒度。正是这些升级操作，为锁的自动降级提供了可能。这里不再多述，详情可参考 `upgrade_shared_lock()` 方法和其调用上下文。

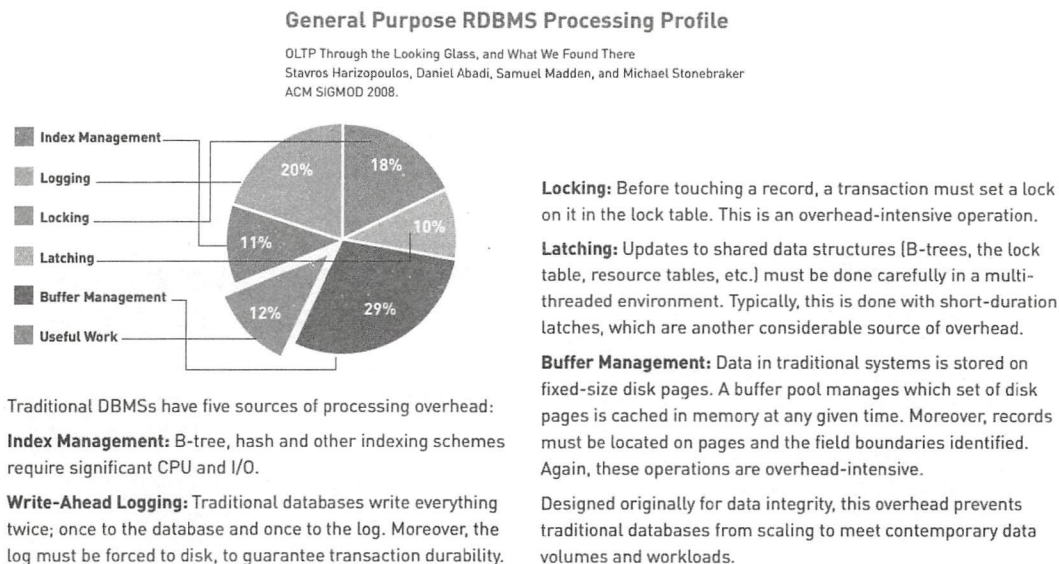


图 11-6 数据库引擎中功能模块资源损耗图<sup>Ⓔ</sup>

## 7. 元数据锁的回滚

在一个 SQL 执行的过程中，一个完整的事务，包括了很多不同的子过程，这些子过程可能会施加很多不同的锁。当某个过程执行失败的时候，事务要回滚，通常情况下锁是在回滚阶段才释放的。但是，MySQL 提供的元数据锁，却提供了“即时释放”的功能，这个功能是通过 `MDL_savepoint` 这个类和相关的一些操作配合完成，即发生错误立即释放而不是等到调用系统级的回滚（参见 10.3.5 节）才去释放锁。这样做的好处显而易见：锁能被及时的释放，有效提高并发度。

```
/**Savepoint for MDL context. //注意，不是所有的元数据锁可以被“即时释放”的，如
“explicit duration”类型
Doesn't include metadata locks with explicit duration as they are not released
during rollback to savepoint.*/
class MDL_savepoint
{
public:
    MDL_savepoint() {};
```

```
private:
    MDL_savepoint(MDL_ticket *stmt_ticket, MDL_ticket *trans_ticket)
        : m_stmt_ticket(stmt_ticket), m_trans_ticket(trans_ticket){}
```

<sup>Ⓔ</sup> 引自：[https://downloads.voltdb.com/datasheets\\_collateral/technical\\_overview.pdf](https://downloads.voltdb.com/datasheets_collateral/technical_overview.pdf).



```

    friend class MDL_context;
private:
    /**Pointer to last lock with statement duration which was taken before
    creation of savepoint.*/
    MDL_ticket *m_stmt_ticket; //statement duration类可以被及时释放
    /**Pointer to last lock with transaction duration which was taken before
    creation of savepoint.*/
    MDL_ticket *m_trans_ticket; //transaction duration类可以被及时释放
};

```

下面是 MDL\_savepoint 的一个使用场景，其他都类似，逻辑简单，不再过多描述。

```

bool open_tables_for_query(THD *thd, TABLE_LIST *tables, uint flags)
{
    DML_prelocking_strategy prelocking_strategy;
    MDL_savepoint mdl_savepoint= thd->mdl_context.mdl_savepoint();
    //记录下本过程开始前的锁的情况（记录本代码段之前的锁是谁）
    ...

    if (open_tables(thd, &tables, &thd->lex->table_count, flags, &prelocking_
    strategy)) //打开表
        goto end; //打开表失败，表示有异常，需要做异常处理

    DEBUG_RETURN(0);
end: //有异常，需要关闭表，释放这个过程中施加的锁
    ...

    close_thread_tables(thd);
    /* Don't keep locks for a failed statement. */
    thd->mdl_context.rollback_to_savepoint(mdl_savepoint);
    //回滚，就是调用MDL_context::rollback_to_savepoint()释放这个过程中施加的锁

    DEBUG_RETURN(TRUE); /* purecov: inspected */
}

```

对于元数据的回滚，实现和使用方式较为简单，其中的关键之处，在于使用的时机。一个确定的原则是：加锁前通过 `mdl_savepoint()` 获取当前锁系统的状态，遇到异常，及时通过 `rollback_to_savepoint()` 完成锁资源的释放。

## 8. 防止锁饥饿

如果元数据锁获取不上，想要获取锁的会话便会处于阻塞等待状态而不是忙式等待（即不进入等待状态）。这是 MySQL 对于等待的锁的处理方式。

在申请新锁时，除了和已经获得的锁进行锁的相容性判断外，还要和等待的锁进行优先级判断（MDL\_lock\_strategy 定义的 `m_waiting_incompatible` 二维数组中的第一个数组是用于等待的锁与新申请的锁进行优先级判断的），这样，可能会导致优先级低的锁对应的会话处于长时间的等待或者总是获取不到锁，所以产生了“锁饥饿（starvation）”现象。

MySQL 对于元数据锁的管理方式能够产生锁饥饿现象，但是 MySQL 也提供了相应的方式来一定程度地消除锁饥饿现象。MDL\_lock\_strategy 定义的 m\_waiting\_incompatible 二维数组中的第二、第三、第四个数组，就是为了防止锁饥饿而设计。

MySQL 元数据锁管理模块防止锁饥饿的基本思路，是通过为一些低优先级类型的锁设立计数器，如果计数器达到一定次数（max\_write\_lock\_count 设定最大值），则优先处理低优先级的锁对应的任务并把计数器清零（之后继续重新累计）。

那么，哪些类型的锁属于低优先级的需要使用防止锁饥饿的措施呢？

首先，MySQL 定义了两种类别，分别称为“hog”和“piglet”<sup>①</sup>。

其中，hog 是如下三种粒度的锁：

```
static const bitmap_t MDL_OBJECT_HOG_LOCK_TYPES=
    (MDL_BIT(MDL_SHARED_NO_WRITE) |
     MDL_BIT(MDL_SHARED_NO_READ_WRITE) |
     MDL_BIT(MDL_EXCLUSIVE));
```

而 piglet 是 MDL\_SHARED\_WRITE 锁。

其次，MySQL 定义了计数器的计数规则（计数也是有特定条件的），方式如下：

```
/** If we just have granted a lock of "piglet" or "hog" type and there are pending
lower priority locks,
increase the appropriate counter.
If this counter now exceeds the max_write_lock_count threshold, switch priority
matrice for the MDL_lock object.
@returns true - if priority matrice has been changed, false - otherwise.*/
bool count_piglets_and_hogs(enum_mdll_type type) //注意type对应的是可以被授予的锁
{...
    if ((MDL_BIT(type) & MDL_OBJECT_HOG_LOCK_TYPES) != 0) //hog类别的锁
    {
        if (m_waiting.bitmap() & ~MDL_OBJECT_HOG_LOCK_TYPES)
            //hog类别的锁不处于等待状态
        {
            m_hog_lock_count++; //hog类别的锁不处于等待状态则计数，处于等待状态的不算
            (hog类的锁还没有获取)
            if (switch_incompatible_waiting_types_bitmap_if_needed())
                return true;
        }
    }
    else if (type == MDL_SHARED_WRITE) //piglet类别的锁
    {
        if (m_waiting.bitmap() & MDL_BIT(MDL_SHARED_READ_ONLY))
        {
```

① 注意，这里所指的锁饥饿，是对于object类型的锁而言的。

对于scoped类型的锁，MySQL有注释如此说：In scoped locks, only IX lock request would starve because of X/S. But that is practically a very rare case. So we don't apply the max\_write\_lock\_count limit to them.



```

        m_piglet_lock_count++; //申请piglet类别的锁，而处于等待状态的锁包含有
        MDL_SHARED_READ_ONLY类型的锁才计数
        if (switch_incompatible_waiting_types_bitmap_if_needed())
            //返回值为TRUE，表示计数超过最大值，需要切换给低优先级的锁了
            return true;
    }
}
return false;
}

```

第三，除了技术规则外，需要观察 `count_piglets_and_hogs()` 被调用的上下文（被 `MDL_context::try_acquire_lock_impl()` 和 `MDL_lock::reschedule_waiters()` 调用），以 `MDL_context::try_acquire_lock_impl()` 的部分代码段为例，我们观察 `count_piglets_and_hogs()` 的入口参数，是一个可以被授予的锁请求，所以计数器一定是一个可被授予的锁（例如：此时对于 `m_piglet_lock_count` 计数器可以理解为“Number of times high priority, "piglet" lock requests (SW) have been granted while locks requests with lower priority (SRO) were waiting”，实则这是 MySQL 自身的注释，正好是上面的代码分析中对 `piglet` 类别的代码逻辑）与等待的锁之间关系。

```

if (lock->can_grant_lock(mdl_request->type, this))
//锁请求可以被授予，所以可能是高优先级的
{
    ...
    if (lock->is_affected_by_max_write_lock_count())
    {
        ...
        if (lock->count_piglets_and_hogs(mdl_request->type))
            //可以被授予的锁请求作为入口参数
            lock->reschedule_waiters(); //对于piglet和hog能执行到这里时，其返回值为
            TRUE，则mdl_request->type一定是高优先级的
            //且已经超过计数最大值，意味着需要给低优先级
            锁以被授予的机会
            //所以要执行reschedule_waiters()进行对等
            待者的调度，这就避免了锁饥饿
    }
    ...
}
else
    *out_ticket= ticket;

```

第四，等待的锁获得被授予锁的机会发生在 `MDL_lock::reschedule_waiters()` 里，如下：

```

/** Determine waiting contexts which requests for the lock can be satisfied, grant
lock to them and wake them up.
@note Together with MDL_lock::add_ticket() this method implements fair
scheduling among requests with the same priority.

```



```

        It tries to grant lock from the head of waiters list, while add_ticket()
        adds new requests to the back of this list. */
void MDL_lock::reschedule_waiters()
{...
    MDL_lock::Ticket_iterator it(m_waiting); // m_waiting队列要被迭代
    ...
    /*
        Find the first (and hence the oldest) waiting request which
        // m_waiting队列中第一个是最老的
        can be satisfied (taking into account priority). Grant lock to it.
        Repeat the process for the remainder of waiters.
        Note we don't need to re-start iteration from the head of the list after
        satisfying the first suitable request
        as in our case all compatible types of requests have the same priority.
    */
    while ((ticket= it++))
    {
        if (can_grant_lock(ticket->get_type(), ticket->get_ctx())) //满足授予锁的条件
        {
            if (! ticket->get_ctx()->m_wait.set_status(MDL_wait::GRANTED))
                //又能设置为被授予, 则不再等待了
            {...
                m_waiting.remove_ticket(ticket); //等待队列去掉可被授予的低优先级的锁请求
                m_granted.add_ticket(ticket);    //列入被授予锁的队列

                if (is_affected_by_max_write_lock_count())
                    //低优先级的锁请求被授予 (进入本方法即表明了这一点)
                {
                    if (count_piglets_and_hogs(ticket->get_type()))
                        //但只针对piglet和hog类型
                    {
                        /*
                            Switch of priority matrice might have unblocked some lower-prio
                            locks which are still compatible with the lock type
                            we just have
                            granted (for example, when we grant SNW lock and
                            there are pending
                            requests of SR type). Restart iteration to wake them
                            up, otherwise
                            we might get deadlocks.
                        */
                        it.rewind();
                        continue;
                    }
                }
            }
        }
    }
}

```



```

...
}
}

if (is_affected_by_max_write_lock_count())
//低优先级的锁请求存在，等待队列中又不包括低优先级的锁请求，则上面所述的计数器重新计数
{...
    if (m_current_waiting_incompatible_idx == 3)
        //2个计数器都达到了上限，计数器重新计数
        {...
            //等待队列中不再包括低优先级的锁请求（之前的while循环已经把等待队列遍历处理过）
            if ((m_waiting.bitmap() & ~(MDL_OBJECT_HOG_LOCK_TYPES |
                                         MDL_BIT(MDL_SHARED_WRITE) | MDL_BIT(MDL_
                                         SHARED_WRITE_LOW_PRIO))) == 0)
            {
                m_piglet_lock_count= 0; //计数器重新计数
                m_hog_lock_count= 0;
                m_current_waiting_incompatible_idx= 0;
            }
        }
    else //2个计数器之一达到了上限，相应的计数器重新计数
    {
        if ((m_waiting.bitmap() & ~MDL_OBJECT_HOG_LOCK_TYPES) == 0)
        {
            m_hog_lock_count= 0;
            m_current_waiting_incompatible_idx&= ~2;
        }
        if ((m_waiting.bitmap() & MDL_BIT(MDL_SHARED_READ_ONLY)) == 0)
        {
            m_piglet_lock_count= 0;
            m_current_waiting_incompatible_idx&= ~1;
        }
    }
}
}
}

```

最后，注意，`m_hog_lock_count` 和 `m_piglet_lock_count` 这两个计数器是属于 `MDL_lock` 对象的，这表明对于每一个数据库中的对象，在这个对象被各种粒度的锁申请时，防止锁饥饿一定是因该对象上存在多个、多种、各样粒度的锁对此对象竞争，才需要消除面向此对象的锁饥饿现象。

### 11.4.3 死锁处理

死锁处理，使用等待图来找出图中的环，然后根据一定的规则，确定环上哪个会话应该作为受害者。

## 1. 死锁检测的入口

非行级锁的死锁检测，是在元数据锁的上下文空间内进行的，这个空间包括了一个用户会话的所有的锁的相关情况。所以，死锁检测也是在MDL\_context这个空间内进行的。死锁检测的入口方法是MDL\_context这个空间的find\_deadlock()。

```
//找出死锁，就是找出一个环。找出以后，就需要打破这个环，从环上摘除一个点，就是受害者点，此时有两种情况：
//一是受害者就是自己；二是受害者上没有访问者
void MDL_context::find_deadlock() //Try to find a deadlock. This function
produces no errors.
{
    while (1)
    {
        /*The fact that we use fresh instance of gvisitor for eachsearch
        performed by find_deadlock() below is important,
        the code responsible for victim selection relies on this. */
        Deadlock_detection_visitor dvisitor(this); //this是当前的MDL_context对象，申请锁者。
        //从申请锁者出发，找出受害者。在等待图中，申请锁者就是出发点，之后从这个点为起始点，深度遍历等待
        //图，当再次访问到
        //起始点时，就表明存在死锁（有环存在）。下文对MDL_lock::visit_subgraph()方法的分析要用到这一点。
        MDL_context *victim; //受害者也是一个MDL_context对象

        //等待图上不存在来访者，则表明没有死锁。退出循环的条件之一。如果存在死锁则找出受害者，供
        get_victim()直接获取受害者
        if (! visit_subgraph(&dvisitor))
        {
            break; /* No deadlocks are found! */
        }

        victim= dvisitor.get_victim(); //否则，存在死锁，需要从当前对象死锁检测访问者
        dvisitor出发找出受害者

        /*Failure to change status of the victim is OK as it meansthat the victim
        has received some other message
        and is about to stop its waiting/to break deadlock loop.
        Even when the initiator of the deadlock search is chosen the victim,
        we need to set the respective wait
        result in order to "close" it for any attempt to schedule the request.
        This is needed to avoid a possible race during cleanup in case when
        the lock request on which
        the context was waiting is concurrently satisfied. */
        //这一点很重要，被选为受害者的标志就是设置状态为“MDL_wait::VICTIM”。在受害者对应
        //的会话内，在获取锁的时候，
        //即在MDL_context内执行 MDL_context::acquire_lock()时，就会根据wait_status的值是
        VICTIM执行
        //“my_error(ER_LOCK_DEADLOCK, MYF(0));”，让受害者对应的会话给用户报告发生死锁的错误信息。
```



```

        (void) victim->m_wait.set_status(MDL_wait::VICTIM); //所以这个标志的设定相当
        于并发会话的一个消息通知
        victim->unlock_deadlock_victim();

        if (victim == this) //来访者(本会话)被选为了受害者,是自己造成了死锁。这是退出循
        环的另外一个条件
            break;
        /*After adding a new edge to the waiting graph we found that it creates
        a loop (i.e. there is a deadlock).
        We decided to destroy this loop by removing an edge, but not the one
        that we added.
        Since this doesn't guarantee that all loops created by addition of the
        new edge are destroyed,
        we have to repeat the search.*/ //解除死锁的思路,就是找出环中的受害者,破
        坏形成环
    }
}

```

## 2. 死锁检测的方法

MDL\_context::find\_deadlock() 调用了 MDL\_context::visit\_subgraph() 来确认是否有环存在。判断依据是“同一个节点被访问两次即有环存在”。

```

/**
    A fragment of recursive traversal of the wait-for graph of MDL contexts in the
    server in search for deadlocks.
    Assume this MDL context is a node in the wait-for graph, and direct the
    visitor to all adjacent nodes.
    As long as the starting node is remembered in the visitor, a deadlock is found
    when the same node is visited twice.
    One MDL context is connected to another in the wait-for graph if it waits on a resource
    that is held by the other context.

    @retval TRUE    A deadlock is found. A pointer to deadlock victim is saved in the visitor.
    @retval FALSE*/
    bool MDL_context::visit_subgraph(MDL_wait_for_graph_visitor *gvisitor)
    //同一个节点被访问过2次,就表明有环存在
{
    bool result= FALSE;
    mysql_prlock_rdlock(&m_LOCK_waiting_for);
    //不存在锁等待则不用继续搜索
    if (m_waiting_for) //在 acquire_lock() 中调用 will_wait_for(ticket) 的原因就在于此:
    为 m_waiting_for 赋值, 存在锁等待
        result= m_waiting_for->accept_visitor(gvisitor); //表示等待图的 m_waiting_
        for 也是一个 MDL_ticket 对象
    mysql_prlock_unlock(&m_LOCK_waiting_for);
}

```

```

    return result;
}

```

而 MDL\_ticket::accept\_visitor() 又是通过 m\_lock->visit\_subgraph() 实现是否有环的判断。

```

bool MDL_ticket::accept_visitor(MDL_wait_for_graph_visitor *gvisitor)
{
    //转到ticket所栖息的MDL_lock中搜索, 因为死锁只会发生在实体的锁对象间
    return m_lock->visit_subgraph(this, gvisitor); //m_lock是注册在ticket上的,
    表示ticket所依附的MDL_lock是谁
}

```

所以, 我们重点来看 MDL\_lock::visit\_subgraph()。

```

/**
 * A fragment of recursive traversal of the wait-for graph in search for deadlocks.
 * Direct the deadlock visitor to all contexts that own the lock the current node
 * in the wait-for graph is waiting for.
 * As long as the initial node is remembered in the visitor, a deadlock is found
 * when the same node is seen twice.*/
bool MDL_lock::visit_subgraph(MDL_ticket *waiting_ticket, MDL_wait_for_graph_
visitor *gvisitor)
{
    //waiting_ticket是MDL_context::acquire_lock()里不能被授予锁的处于等待状态的ticket
    对象, 即新进入等待图的对象
    //gvisitor是死锁检测的发起者, 即锁的申请者, 在等待图中为第一个要遍历的节点。即waiting_
    ticket所在的MDL_context。
    MDL_ticket *ticket;
    //进入本方法前在MDL_context::acquire_lock()里src_ctx->m_wait值是MDL_wait::EMPTY
    MDL_context *src_ctx= waiting_ticket->get_ctx();
    bool result= TRUE; //如果result为TRUE, 表示有死锁; 如下有四种情况⊖, 如果存在死锁, 则
    不修改result的值

    mysql_prlock_rdlock(&m_rwlock);

    /*Iterators must be initialized after taking a read lock.
    // “fast path” 锁之间互斥不会产生死锁, 故不考虑
    Note that MDL_ticket's which correspond to lock requests satisfied on "fast
    path" are not present in m_granted list
    and thus corresponding edges are missing from wait-for graph.
    It is OK since contexts with "fast path" tickets are not allowed to wait
    for any resource
    (they have to convert "fast path" tickets to normal tickets first) and thus cannot
    participate in deadlock.
    @sa MDL_context::will_wait_for().*/
    Ticket_iterator granted_it(m_granted); //m_granted是MDL_lock对象上已经被授予的锁列表
    Ticket_iterator waiting_it(m_waiting); //m_waiting是MDL_lock对象上处于等待状态的锁列表

    /*MDL_lock's waiting and granted queues and MDL_context::m_waiting_for member

```

⊖ 为什么需要对这四种情况进行判断, 请参见这个方法的源码分析之后的下文。



are updated by different threads

when the lock is granted (see MDL\_context::acquire\_lock() and MDL\_lock::reschedule\_waiters()).

As a result, here we may encounter a situation when MDL\_lock data already reflects the fact

that the lock was granted but m\_waiting\_for member has not been updated yet.

For example, imagine that:

thread1: Owns SNW lock on table t1.

thread2: Attempts to acquire SW lock on t1, but sees an active SNW lock.

Thus adds the ticket to the waiting queue and sets m\_waiting\_for to point to the ticket.

thread1: Releases SNW lock, updates MDL\_lock object to grant SW lock to thread2 (moves the ticket for

SW from waiting to the active queue). Attempts to acquire a new SNW lock on t1,

sees an active SW lock (since it is present in the active queue), adds ticket for SNW lock to the waiting queue, sets m\_waiting\_for to point to this ticket.

At this point deadlock detection algorithm run by thread1 will see that:

- Thread1 waits for SNW lock on t1 (since m\_waiting\_for is set).
- SNW lock is not granted, because it conflicts with active SW lock owned by thread 2 (since ticket for SW is present in granted queue).
- Thread2 waits for SW lock (since its m\_waiting\_for has not been updated yet!).
- SW lock is not granted because there is pending SNW lock from thread1.

Therefore deadlock should exist [sic!].

To avoid detection of such false deadlocks we need to check the "actual" status of the ticket being waited for,

before analyzing its blockers. We do this by checking the wait status of the context which is waiting for it.

To avoid races this has to be done under protection of MDL\_lock::m\_rwlock lock.

\*/

```
//进入本方法前在MDL_context::acquire_lock()里src_ctx->m_wait值是MDL_wait::EMPTY
if (src_ctx->m_wait.get_status() != MDL_wait::EMPTY)
{
    result= FALSE;//现在, 值发生了变化, 表明被别的会话修改了状态(发生状态设置的情况参见
    图11-7), 但没有死锁
    goto end;
}
```

```
/*To avoid visiting nodes which were already marked as victims of deadlock
detection (or whose requests
were already satisfied) We enter the node only after peeking at its wait status.
This is necessary to avoid active waiting in a situation when previous
searches for a deadlock
already selected the node we're about to enter as a victim (see the
```



```

comment in MDL_context::find_deadlock()
for explanation why several searches can be performed for the same wait).
There is no guarantee that the node isn't chosen a victim while we are
visiting it but this is OK:
in the worst case we might do some extra work and one more context might
be chosen as a victim.*/
if (gvisitor->enter_node(src_ctx)) //如果搜索深度超过MAX_SEARCH_DEPTH定义的32层,
则认为发生了死锁。另外,存在一个根据权重选受害者的过程,权重小的被选为受害者。相关上下文调用关系
参见图11-8
goto end; //执行此行代码,表明确认有死锁,所以result的TRUE值不会被修改为FLASE

/*We do a breadth-first search first -- that is, inspect all edges of the
current node,
and only then follow up to the next node. In workloads that involve wait-
for graph loops
this has proven to be a more efficient strategy [citation missing].*/
while ((ticket= granted_it++)) //深度遍历,每次迭代访问初始ticket的next_in_lock
{
    /* Filter out edges that point to the same node. */
    if (ticket->get_ctx() != src_ctx && //与申请锁者(waiting_ticket所在会话)
        不是同一个会话,同一个会话则不必要进行死锁检测了(处于同一个事务无并发竞争)
        ticket->is_incompatible_when_granted(waiting_ticket->get_type()) &&
        //等待申请的锁与已经授予的不相容
        gvisitor->inspect_edge(ticket->get_ctx())) //已经授予锁的节点就是“发起
        者”,表明第二次访问到了自己,有死锁
    {
        goto end_leave_node;
    }
}

while ((ticket= waiting_it++)) //深度遍历MDL_lock上的处于等待的锁,查新申请的锁是
否构成死锁
{
    /* Filter out edges that point to the same node. */
    if (ticket->get_ctx() != src_ctx && //与申请锁者(waiting_ticket所在会话)不
        是同一个会话
        ticket->is_incompatible_when_waiting(waiting_ticket->get_type()) &&
        //等待申请的锁与早已处于等待锁的不相容
        gvisitor->inspect_edge(ticket->get_ctx())) //已经授予锁的节点就是“发起
        者”,表明第二次访问到了自己,有死锁
    {
        goto end_leave_node;
    }
}

/* Recurse and inspect all adjacent nodes. */
granted_it.rewind(); //重置,重新遍历
while ((ticket= granted_it++))
{

```



```

    if (ticket->get_ctx() != src_ctx &&
        //与申请锁者（waiting_ticket所在会话）不是同一个会话
        ticket->is_incompatible_when_granted(waiting_ticket->get_type()) &&
        //等待申请的锁与已经授予的不相容
        ticket->get_ctx()->visit_subgraph(gvisitor))
        //水平访问每一个已经得到锁的所在的锁上下文空间（MDL_context）
        //中的已经授予的锁和等待的锁，在此空间中递归搜索（注意是递归）。递归之后的结果
        返回true，则有死锁
    {
        goto end_leave_node;
    }
}

waiting_it.rewind(); //重置，重新遍历
while ((ticket= waiting_it++))
{
    if (ticket->get_ctx() != src_ctx &&
        //与申请锁者（waiting_ticket所在会话）不是同一个会话
        ticket->is_incompatible_when_waiting(waiting_ticket->get_type()) &&
        //等待申请的锁与早已处于等待锁的不相容
        ticket->get_ctx()->visit_subgraph(gvisitor))
        //水平访问每一个申请锁者的所在的锁上下文空间（MDL_context）
        //中的已经授予的锁和等待的锁，在此空间中递归搜索（注意是递归）。递归之后的结果返回
        true，则有死锁
    {
        goto end_leave_node;
    }
}

result= FALSE; //否则，没有发现死锁

end_leave_node:
    gvisitor->leave_node(src_ctx);

end:
    mysql_prlock_unlock(&m_rwlock);
    return result;
}

```

在前文，我们分析了死锁检测的代码，需要明白死锁检测的算法为什么需要有上述四种搜索方式。

这需要从MDL\_key这种对象说起。MDL\_key定义了多种key，除了GLOBAL和COMMIT类是全局且唯一的，其他类型的key可以都有多个对象存在。而MDL\_key本身所起的作用就是划分锁的类别。

这就是说，对于其他的key类，可以有多个。且每一个里面，一定是同一类的锁。所以才有了MDL\_lock，用MDL\_lock把同一类别的相关的锁聚拢。这里的同一类别，不是一个很大的类别，而是一个用三元组标识的一个具体对象，格式如下：

```
<mdl_namespace>+<database name>+<table name>
```

在这个格式中，首先是 key 值，然后是对象所属的数据库名称和对象名。例如，在 db01 这个数据库中存在表名为 user01、user02 的两张表，则会生成两个 MDL\_lock 对象，分别是：

```
{3,db01,user01}和{3,db01,user02}
```

第一个 3 是 key 值为 MDL\_key::TABLE 所对应的值，后两个分别是数据库名和表名。所以，每个对象都对应着一个 MDL\_lock。即每个 MDL\_lock 表示一个具体的数据库对象上存在有哪些已经授予的锁，有哪些锁在等待授予。

所以加锁成功，一定是在一个具体的对象（数据库对象和 MDL\_lock 通过上述三元组建立映射）上授予了一个“想”拥有锁的凭证 ticket，然后将其存入到 MDL\_lock 的“Ticket\_list m\_granted”中才算是加锁成功完成；加锁不成功，则把“想”拥有锁的凭证 ticket 存入到 MDL\_lock 的“Ticket\_list m\_waiting”以表示哪些锁处于等待状态。

当在一个对象上判断的时候，就一定要看这个对象上：

❑ 已经被授予的锁和新申请的锁是否构成环（第一个循环）。

❑ 已经处于等待的锁和新申请的锁是否构成环（第二个循环）。

❑ 已经被授予的锁所在的空間（MDL\_context）的所有已经授予和等待的锁与新申请的锁是否构成环（第三个循环，在这个循环中会重复前两步，只是切换了锁上下文空间）。

❑ 已经处于等待的锁和新申请的锁是否构成环（第四个循环，在这个循环中也会重复第一和第二步，只是切换了锁上下文空间）。

经过这四个循环，把所有的锁遍历了一遍，死锁检测才算完整。但是，遍历的代价会很高，如果并发竞争激烈，存在很多会话，死锁检测的效率就可能成为一个瓶颈（MySQL 给出的一个改进方案，参见 11.8.2 节）。

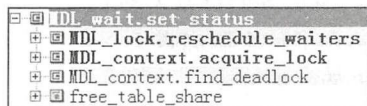


图 11-7 锁等待状态设置图

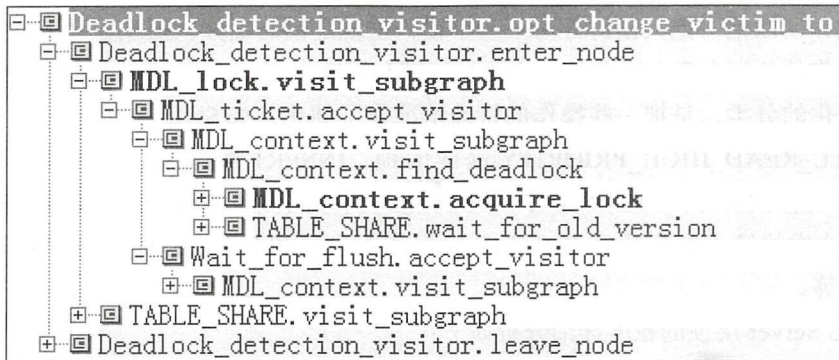


图 11-8 死锁检测调用关系上下文图



## 11.5 SQL语义定义锁

MySQL 整体是一个插件式的结构，上层是一个 Server，提供词法分析、语法分析、SQL 优化和执行器的执行框架（包括代码中 src 目录下的文件），其他部分是以插件的形式插入到 MySQL Server 中。如 InnoDB 就是一个负责事务和存储的插件（包括 storage\innobase 目录下的文件）。

对于事务的处理，并不是简单地交由各个插件独自处理，而是在 MySQL Server 层开始，就做了宏观的规定，这样的规定落实在了 thr\_lock.h、thr\_lock.c、handler.cc、handler.h 这些文件中。

对于数据库引擎，怎么才能知道在什么样的情况下应该施加什么样的锁？这对于基于封锁机制的并发控制技术的实现而言，是一个重要的问题，是需要在设计阶段就要考虑好的问题。

所以尽管本书以较大篇幅讨论 InnoDB 的事务处理和并发控制技术，但是也不能忽略 MySQL Server 层对锁的定义和使用。从 MySQL Server 层的角度看待锁的内容，实际上是站在了一个制高点上，可以高屋建瓴地从更高的角度思考数据库系统基于锁的并发控制技术的实现方式。

而 InnoDB 对于记录锁和元数据锁等的并发控制实现技术，实则是承接于 MySQL Server 层对加锁语义的定义所进行的底层实现，这在 11.3 和 11.4 节进行了详细的讨论。本节将从更宏观的角度，讨论 MySQL/InnoDB 关系数据库中对各种 SQL 语义的封锁机制的整体实现方式（不再局限于 11.4 节中的元数据锁）。

### 11.5.1 锁的粒度

在 thr\_lock.h 文件中，定义了 MySQL Server 层锁的粒度（更多人把这些称为锁的类型，而作者是从加锁的意义的角度上进行划分的，封锁思维可以用一把大锁锁住整个事务，但并发性很低，所以把一把大锁拆分为一些小锁，这就是锁的细分、锁从粒度上的细分）。这些锁是从操作的角度宏观上区分的，主要是读锁和写锁。

更为细化的分类，是把一些操作的特殊情况作了细化，如 SQL 语句的语义上加了优先级则引出 TL\_READ\_HIGH\_PRIORITY 这样的锁，INSERT 操作被底层的锁（LOCK\_X+LOCK\_GAP 或者 LOCK\_ORDINARY）机制允许并发引出 TL\_WRITE\_CONCURRENT\_DEFAULT 这样的锁（这表明 MySQL Server 层作为上层和 InnoDB 下层在实现锁设计时互相影响）等。

MySQL Server 层锁的粒度详细分析如下：

```
enum thr_lock_type { //MySQL Server层为各种SQL语句的并发操作引发的各种锁
    TL_IGNORE=-1, //不需要加锁
    TL_UNLOCK,    /* UNLOCK ANY LOCK */
```

```

TL_READ_DEFAULT, /*Parser only! At open tables() becomes TL_READ or TL_READ_
NO_INSERT depending on the binary log format (SBR/RBR)
                                and on the table category (log table). Used for
tables that are read by statements which modify tables.*/
TL_READ, /* Read lock */ //读锁, 如“LOCK TABLE bluesea READ;”施加读锁,
但优先级低
TL_READ_WITH_SHARED_LOCKS, //SELECT语句中带有“LOCK IN SHARE MODE”子句所施加的锁
TL_READ_HIGH_PRIORITY, /* High prior. than TL WRITE. Allow concurrent insert */
//SELECT语句中带有“HIGH_PRIORITY”子句所施加的锁
TL_READ_NO_INSERT, /* READ, Don't allow concurrent insert */
//适用于“FLUSH TABLES table_list [WITH READ LOCK]”命令
TL_WRITE_ALLOW_WRITE, /* Write lock, but allow other threads to read / write.
Used by BDB tables in MySQL to mark
                                that someone is reading/writing to the table.*/
//一些引擎如ndbcluster允许并发的写, 不用于InnoDB
TL_WRITE_CONCURRENT_DEFAULT, // parser only! Late bound low_priority_flag. At
open_tables() becomes thd->insert_lock_default.
TL_WRITE_CONCURRENT_INSERT, //WRITE lock used by concurrent insert. Will
allow READ, if one could use concurrent insert on table.
//允许多个LOAD或INSERT并发操作
TL_WRITE_DEFAULT, //parser only! Late bound low_priority flag. At open_
tables() becomes thd->update_lock_default.
TL_WRITE_LOW_PRIORITY, /* WRITE lock that has lower priority than TL_READ */
//写锁, 如“LOCK TABLE bluesea LOW_PRIORITY WRITE”施加写锁
TL_WRITE, /* Normal WRITE lock */ //常规的写锁
TL_WRITE_ONLY, /* Abort new lock request with an error */
};

```

从上面的分析中可以看出, 锁带有优先级, MySQL 对于这些锁的优先级从高到低的优先次序如下:

```

WRITE_ALLOW_WRITE > WRITE_CONCURRENT_INSERT > WRITE_LOW_PRIORITY > READ > WRITE > READ_
HIGH_PRIORITY > WRITE_ONLY

```

而这些锁, 是在 SQL 语句的语义下被定义而出的, 如下面所列举的两个 SQL 语句, LOCK 语句和 SELECT 语句, 每条 SQL 语句的子句分别定义了不同类型的封锁方式, 从而使得数据库引擎在实现时需提供相应的锁粒度。

例如, LOCK 语句的语法定义 (黑体字确定了多种粒度的锁) 如下, 其中即有与上述的 thr\_lock\_type 所对应的类型:

```

LOCK TABLES      //为指定的表加锁
tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
//可以施加的锁对应上面的thr_lock_type中的锁
[, tbl_name [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...
UNLOCK TABLES    //为指定的表解锁

```



SELECT 语句的语法定义 (黑体字确定了多种粒度的锁):

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
    [HIGH_PRIORITY] //使得锁带有高优先级
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr, ...
    [INTO OUTFILE 'file_name' export_options
     | INTO DUMPFILE 'file_name']
    [FROM table_references
    [WHERE where_definition]
    [GROUP BY {col_name | expr | position}
     [ASC | DESC], ... [WITH ROLLUP]]
    [HAVING where_definition]
    [ORDER BY {col_name | expr | position}
     [ASC | DESC] , ...]
    [LIMIT [{offset,} row_count | row_count OFFSET offset]]
    [PROCEDURE procedure_name(argument_list)]
    [FOR UPDATE | LOCK IN SHARE MODE] //不同的显式的加锁粒度
```

### 11.5.2 重要的数据结构

在 row0mysql.h 文件中, 定义了一个重要的结构体, 存放了一个表的元组操作相关的一些重要信息, InnoDB 的注释说 “A struct for (sometimes lazily) prebuilt structures in an Innobase tablehandle used within MySQL; these are used to save CPU time.”, 确实这个结构体能够起到一个 “cache” 的作用, 很多变量不用重新生成, 但其主要的作用, 还是汇集 “row” 上的相关操作, 其中, 与锁相关的内容如下:

```
struct row_prebuilt_t {...
    uint select_lock_type; /*!< LOCK_NONE, LOCK_S, or LOCK_X */
    //在记录上要施加的锁的粒度是LOCK_S或LOCK_X, 或不加锁
    //值源自external_lock()、store_lock()函数, 参见11.5.3节下的“3.接口层的衔接”
    uint stored_select_lock_type; /*!< this field is used to remember the
    original select_lock_type
        that was decided in ha_innodb.cc, ::store_lock(), ::external_
lock(), etc. */
    uint row_read_type; /*!< ROW_READ_WITH_LOCKS if row lock should be the
    obtained for records under an UPDATE or DELETE cursor.
    If innodb_locks_unsafe_for_binlog is TRUE, this can be set to ROW_
READ_TRY_SEMI_CONSISTENT,
    so that if the row under an UPDATE or DELETE cursor was locked by
another transaction,
    InnoDB will resort to reading the last committed value ('semi-
```



```

consistent read').
Then, this field will be set to ROW_READ_DID_SEMI_CONSISTENT
to indicate that.
If the row does not match the WHERE condition, MySQL will invoke
handler::unlock_row() to
clear the flag back to ROW_READ_TRY_SEMI_CONSISTENT and to simply
skip the row.
If the row matches, the next call to row_search_for_mysql() will
lock the row.
This eliminates lock waits in some cases; note that this
breaks serializability. */
uint new_rec_locks; /* !< normally 0; if srv_locks_unsafe_for_binlog
is TRUE or session is using
READ COMMITTED or READ UNCOMMITTED isolation level, set in row_
search_for_mysql() if we set a new
record lock on the secondary or clustered index; this is used in
row_unlock_for_mysql()
when releasing the lock under the cursor if we determine after
retrieving the row
that it does not need to be locked('mini-rollback') */
...}

```

这个结构体中的 `select_lock_type` 存放了操作所应施加的锁的粒度值，其具体的使用方式，参见下一节。另外，需要注意的是，这个结构体隶属于 InnoDB，所以此结构体一旦被使用，则表明已经进入 InnoDB 层。

另外的三个重要的结构体，属于 MySQL Server 层。

```

typedef struct st_mysql_lock
// 定义 MySQL Server 层的锁信息：属于高层信息，包括表和表对应的锁信息
{
    TABLE **table;
    uint table_count, lock_count;
    THR_LOCK_DATA **locks; // 定义 MySQL Server 层的锁信息：表对应的锁信息
} MYSQL_LOCK;

```

线程 /SESSION 中锁的整体相关信息：

```

typedef struct st_thr_lock_data { // 定义 MySQL Server 层的锁信息：表对应的锁信息
    THR_LOCK_INFO *owner; // 属主是哪个线程、即隶属于哪个 SESSION
    struct st_thr_lock_data *next, **prev; // 前后指针
    struct st_thr_lock *lock; // 某个具体的锁，其结构如下
    mysql_cond_t *cond;
    enum thr_lock_type type; // 锁的粒度，属于 MySQL Server 层，上一节详细讨论
    void *status_param; /* Param to status functions */
    void *debug_print_param;
    struct PSI_table *m_psi;
} THR_LOCK_DATA;

```





某个线程 /SESSION 上的锁的管理信息：

```
typedef struct st_thr_lock {
    LIST list;
    mysql_mutex_t mutex;
    struct st_lock_list read_wait;    //等待本线程/SESSION的读等待
    struct st_lock_list read;         //本线程/SESSION的所有读锁
    struct st_lock_list write_wait;   //等待本线程/SESSION的写等待
    struct st_lock_list write;        //本线程/SESSION的所有写锁
    /* write_lock_count is incremented for write locks and reset on read locks */
    ulong write_lock_count;
    uint read_no_write_count;
    void (*get_status)(void*, int); /* When one gets a lock */
    void (*copy_status)(void*, void*);
    void (*update_status)(void*); /* Before release of write */
    void (*restore_status)(void*); /* Before release of read */
    my_bool (*check_status)(void *);
} THR_LOCK;
```

## 11.5.3 InnoDB 对接 MySQL Server

11.3 节详细讨论了 InnoDB 的记录锁和元数据锁等事务锁，这些锁都是 InnoDB 引擎内部对于数据和元数据所施加的锁。为什么要施加这些锁？谁决定施加什么粒度的锁？

这些问题很重要，但却没有在 11.3 节进行讨论。

从 InnoDB 引擎内部的代码入手，逐步追查，我们可以追溯到 MySQL Server 层。为什么能追溯到 MySQL Server 层呢？InnoDB 引擎在并发控制技术方面和 MySQL Server 层之间存在什么关系，这正是本节将要讨论的问题。

### 1. SQL 语义定义了加锁的粒度

在 sql\_lex.h 文件中，定义了 Parser 阶段所要确定的两个变量的值，这两个值就是下面所描述的 m\_lock\_type 和 m\_mdln\_type。

```
class Yacc_state //词法分析时关键字的值
{
...
    /**
        Type of lock to be used for tables being added to the statement's table
        list in table_factor,
        table_alias_ref, single_multi and table_wild_one rules.
        Statements which use these rules but require lock type different from one
        specified by this member
        have to override it by using st_select_lex::set_lock_for_tables() method.
        //这句描述的情况主要适用与 UPDATE 和 INSERT 语句

        The default value of this member is TL_READ_DEFAULT. The only two cases in
```



```

    which we change it are:
    - When parsing SELECT HIGH_PRIORITY. //参见11.5.1节中SELECT语句的语法
    - Rule for DELETE. In which we use this member to pass information about
      type of lock from delete to single_multi part of rule.

    We should try to avoid introducing new use cases as we would like to get
    rid of this member eventually.
*/
thr_lock_type m_lock_type; //为记录/元组记载锁的粒度(参见11.5.1节)

// The type of requested metadata lock for tables added to the statement table list.
enum_md_type m_md_type; //为表结构/元信息记载锁的粒度
...
void reset() //设置锁的缺省值
{
    yacc_yyss= NULL;
    yacc_yyvs= NULL;
    yacc_yyls= NULL;
    m_lock_type= TL_READ_DEFAULT; //为记录/元组记载锁的粒度对应的缺省值
    m_md_type= MDL_SHARED_READ; //为表结构/元信息记载锁的粒度对应的缺省值
    m_ha_rkey_mode= HA_READ_KEY_EXACT;
}
...}

```

当SQL语句被Parser进行词法分析的时候,根据SQL语句的语义,对m\_lock\_type和m\_md\_type进行直接赋值,或通过add\_table\_to\_list()函数进行赋值。例如:

例1, DROP TABLE 操作发生时,在sql\_yacc.yy文件中定义了其对应的如下代码:

```

drop:
    DROP opt_temporary table_or_tables if_exists
    {
        LEX *lex=Lex;
        lex->sql_command = SQLCOM_DROP_TABLE;
        lex->drop_temporary= $2;
        lex->drop_if_exists= $4;
        YYPS->m_lock_type= TL_UNLOCK; //表被删除,不需要在记录上加锁
        YYPS->m_md_type= MDL_EXCLUSIVE; //表被删除,需要在元数据上加排它锁
    }

```

例2, TRUNCATE TABLE 操作发生时,在sql\_yacc.yy文件中定义了其对应的如下代码:

```

truncate:
    TRUNCATE_SYM opt_table_sym
    {
        LEX* lex= Lex;
        lex->sql_command= SQLCOM_TRUNCATE;
        lex->alter_info.reset();
    }

```





```

YYPS->m_lock_type= TL_WRITE;//表被TRUNCATE,是数据操作,需要在记录上加锁
YYPS->m_md1_type= MDL_EXCLUSIVE;    //表被TRUNCATE,需要在元数据上加排它锁
}

```

例 3, FLUSHTABLE 操作发生时, 在 sql\_yacc.yy 文件中定义了其对应的如下代码:

```

flush_options:
    table_or_tables
    {
        Lex->type|= REFRESH_TABLES;
        //Set type of metadata and table locks forFLUSH TABLES table_list [WITH READ LOCK].
        YYPS->m_lock_type= TL_READ_NO_INSERT;
        YYPS->m_md1_type= MDL_SHARED_HIGH_PRIO;
    }

```

例 4, RENAME TABLE 操作发生时, 在 sql\_yacc.yy 文件中定义了其对应的如下代码, 通过 add\_table\_to\_list() 函数对锁的粒度变量进行赋值:

```

table_to_table: //执行RENAME table_name命令时使用
    table_ident TO_SYM table_ident
    {
        LEX *lex=Lex;
        SELECT_LEX *sl= Select;
        if (!sl->add_table_to_list(lex->thd, $1,NULL,TL_OPTION_UPDATING,
            TL_IGNORE, MDL_EXCLUSIVE) ||
            !sl->add_table_to_list(lex->thd, $3,NULL,TL_OPTION_UPDATING,
            TL_IGNORE, MDL_EXCLUSIVE))
            MYSQL_YYABORT;
    }

```

例 5, SELECT 操作发生时, 在 parse\_tree\_nodes.h 文件中定义了其对应的如下代码:

```

class PT_select_options_and_item_list : public Parse_tree_node
{
...
    virtual bool contextualize(Parse_context *pc)
    {
...
        if (options.query_spec_options &SELECT_HIGH_PRIORITY)
            //如果SQL语句指定“HIGH_PRIORITY”则锁的值为TL_READ_HIGH_PRIORITY
        {
            Yacc_state *yyys= &pc->thd->m_parser_state->m_yacc;
            yyys->m_lock_type= TL_READ_HIGH_PRIORITY;
            yyys->m_md1_type= MDL_SHARED_READ;
        }
...
    }
};

```

例 6, LOCK TABLE 操作发生时, 在 sql\_yacc.yy 文件中定义了其对应的如下代码:



```

table_lock: //LOCK TABLE...
    table_ident opt_table_alias lock_option
    {
        thr_lock_type lock_type= (thr_lock_type) $3; //值由lock_option确定
        enum_md1_type mdl_lock_type;

        if (lock_type >= TL_WRITE_ALLOW_WRITE) //根据记录锁的值再确定元数据的锁的值
        {
            /* LOCK TABLE ... WRITE/LOW_PRIORITY WRITE */
            mdl_lock_type= MDL_SHARED_NO_READ_WRITE;
        }
        else if (lock_type == TL_READ)
        {
            /* LOCK TABLE ... READ LOCAL */
            mdl_lock_type= MDL_SHARED_READ;
        }
        else
        {
            /* LOCK TABLE ... READ */
            mdl_lock_type= MDL_SHARED_READ_ONLY;
        }

        if (!Select->add_table_to_list(YTHD, $1, $2, 0, lock_type, mdl_lock_type))
            //把记录锁值和元数据锁值记载到表对象上
            MYSQL_YABORT;
    }
;

lock_option: //根据SQL语句的子句确定lock_type的值
    READ_SYM { $$= TL_READ_NO_INSERT; }
    //READ_SYM、WRITE_SYM、LOW_PRIORITY WRITE_SYM等是SQL语句中的子句
    | WRITE_SYM { $$= TL_WRITE_DEFAULT; }
    | LOW_PRIORITY WRITE_SYM
    {
        $$= TL_WRITE_LOW_PRIORITY;
        push_deprecated_warn(YTHD, "LOW_PRIORITY WRITE", "WRITE");
    }
    | READ_SYM LOCAL_SYM { $$= TL_READ; }
;

```

例 7, SELECT 操作发生时, 对于 FROM 子句中的派生表(派生表是子查询的一种)在 parse\_tree\_nodes.cc 文件中定义了其对应的如下代码:

```

bool PT_table_factor_select_sym::contextualize(Parse_context *pc)
{...
    // Note that this current select is different from the one above
    pc->select= child;

```





```

pc->select->linkage= DERIVED_TABLE_TYPE; //派生表
pc->select->parsing_place= CTX_SELECT_LIST;
outer_select->parsing_place= CTX_NONE;

Yacc_state *yyyps= &pc->thd->m_parser_state->m_yacc;

if (select_options.query_spec_options &SELECT_HIGH_PRIORITY)
{
    yyyps->m_lock_type= TL_READ_HIGH_PRIORITY;
    yyyps->m_md1_type= MDL_SHARED_READ;
}
...
}

```

从上面的几个例子可以看出，在 Parser 阶段，解析 SQL 语句的时候，已经确定了锁的值，即事务管理器在进行锁的加锁处理时所使用读锁还是写锁或其他粒度的锁，完全是根据 SQL 语句的语义所确定的。

## 2. 确定表对象上的锁类型

上一节讨论了锁的值是由 SQL 语句的语义定义的，其中，除了在词法分析时直接为锁的值赋值外，还如例 4 所示，可调用 `add_table_to_list()` 函数完成锁的赋值。

首先分析 `add_table_to_list()` 函数的实现源码，SQL 语句中指定的锁的值被保存在了表对象的 `lock_type` 变量中：

```

/**
Add a table to list of used tables.

@param table          Table to add //指明表对象
@param alias          alias for table (or null if no alias)
@param table_options  A set of the following bits:
    - TL_OPTION_UPDATING : Table will be updated
    - TL_OPTION_FORCE_INDEX : Force usage of index
    - TL_OPTION_ALIAS : an alias in multi table DELETE
@param lock_type      How table should be locked //表中元组上应当施加的锁，这是对数据的修改
@param mdl_type       Type of metadata lock to acquire on the table.
//表的元数据上应当施加的锁，这是对表结构的修改
@param use_index       List of indexed used in USE INDEX
@param ignore_index    List of indexed used in IGNORE INDEX

@return
    0          Error
    @retval
    \# Pointer to TABLE_LIST element added to the total table list
*/
TABLE_LIST *st_select_lex::add_table_to_list(THD *thd, //对SQL语句做Parser的阶段，

```



```

    即确定应当施加什么样的锁，这是由SQL语句的语义确定的
    Table_ident *table,
    LEX_STRING *alias,
    ulong table_options,
    thr_lock_type lock_type, //参数：表中元组上应当施加的锁，这是对数据的修改
    enum_mdl_type mdl_type, //参数：表的元数据上应当施加的锁，这是对表结构的修改
    List<Index_hint> *index_hints_arg,
    List<String> *partition_names,
    LEX_STRING *option)
{
    TABLE_LIST *ptr;      //表对象
    ...
    ptr->lock_type= lock_type; //表中元组上应当施加的锁，这是对数据的修改
    ...
    // Pure table aliases do not need to be locked:
    if (!MY_TEST(table_options & TL_OPTION_ALIAS))
    {
        MDL_REQUEST_INIT(& ptr->mdl_request,
                          MDL_key::TABLE, ptr->db, ptr->table_name, mdl_type,
//表的元数据上应当施加的锁，这是对表结构的修改
                          MDL_TRANSACTION);
        ...
    }
    ...
}

```

而 lock\_type 的值是由上层函数传入的，如图 11-9 所示，上层函数根据操作的语义给参数传入不同的值，这些值的分析，如表 11-15 所示。

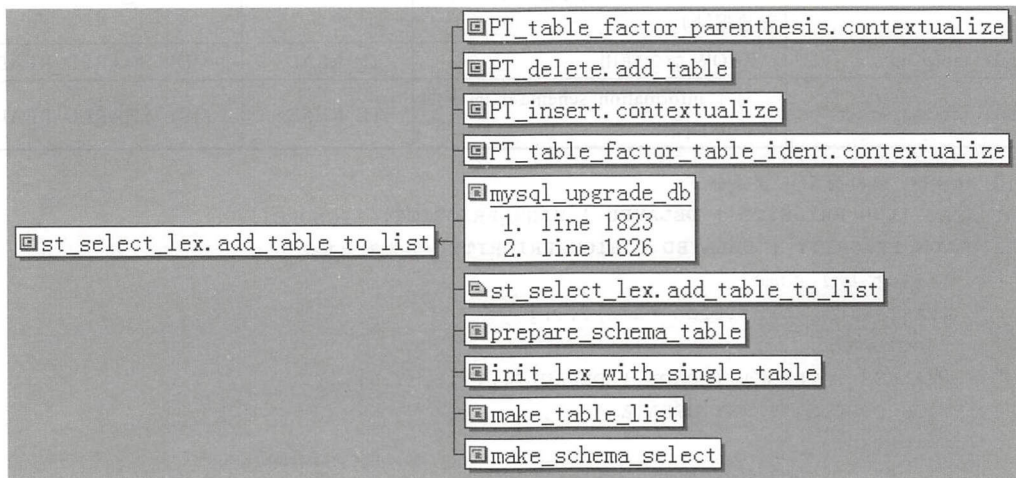


图 11-9 add\_table\_to\_list() 函数调用关系图





表 11-15 调用 add\_table\_to\_list() 函数的上层函数的加锁分析表

函数名	功能	元组锁	元数据锁
PT_table_factor_parenthesis::contextualize	对派生表（FROM 子句中的子查询）的处理	TL_READ	MDL_SHARED_READ
PT_delete::add_table	DELETE 操作	TL_WRITE_ LOW_PRIORITY TL_WRITE_ DEFAULT	MDL_SHARED_ WRITE_LOW_PRIO MDL_SHARED_WRITE
PT_insert::contextualize	INSERT 操作 <sup>①</sup> ，锁的值根据子句的值确定，可能的值有： 没有子句缺省为 TL_WRITE_CONCURRENT_DEFAULT， 子句带有 LOWPRIORITY 时的锁值为 TL_WRITE_LOW_PRIORITY， 子句带有 DELAYED 是锁的值为 TL_WRITE_CONCURRENT_DEFAULT， 子句带有 HIGH PRIORITY 时的锁值为 TL_WRITE		INSERT 操作只对数据，不涉及元数据
mysql_upgrade_db	用于数据升级，ALTER DATABASE 'olddb' UPGRADE DATA DIRECTORY NAME	TL_IGNORE	MDL_EXCLUSIVE
prepare_schema_table	用于 SHOW 或 DESCRIBE table_name 命令，这些命令只从 INFORMATION_SCHEMA 获取信息，所以加锁为 READ 类	TL_READ	MDL_SHARED_READ
init_lex_with_single_table	解析 SQL 语句中表对象的字段所属（即字段属于哪个表，此时，表对象已经在 Parser 阶段被解析完成，因而不需要再考虑表对象上的元组锁和元数据锁）	NULL	NULL
make_table_list	SHOW 命令使用	TL_READ	MDL_SHARED_READ
make_schema_select	对于 information_schema 中的表的 SELECT 操作	TL_READ	MDL_SHARED_READ

① INSERT 语句的 SQL 语法如下：

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
// “LOW_PRIORITY | DELAYED | HIGH_PRIORITY” 与锁的值相关
[INTO] tbl_name
[PARTITION (partition_name,...)]
[(col_name,...)]
{VALUES | VALUE} ({expr | DEFAULT},...), (...),...
[ ON DUPLICATE KEY UPDATE
col_name=expr
[, col_name=expr] ... ]
```

### 3. 接口层的衔接

在 MySQL 中，handler.h 和 handler.c 定义了插件式体系结构的接口层内容，把 MySQL



Server 和诸如 InnoDB 这样的事务、存储引擎衔接起来。

当词法分析和语法分析完成后，锁的值基本确定，那么这些锁的值是怎么传递到 InnoDB 层的呢？

整个过程，贯穿了 MySQL Server 层和 InnoDB 层，他们相互交错，MySQL Server 层通过调用 InnoDB 层的 `ha_innabase::store_lock()`、`ha_innabase::external_lock()`、`ha_innabase::start_stmt()` 函数完成锁的值的传递和赋值调整（调整的含义是 MySQL Server 层和 InnoDB 层的锁的语义不完全相同，某些 MySQL Server 层定义的锁在 InnoDB 层不被支持，如 5.2.3 节提及的 `LOCK TABLES t1,...,tn LOW_PRIORITY WRITE`）。

如下以“`LOCK TABLES...`”语句为例，说明锁从 MySQL Server 层向 InnoDB 层传递的过程。首先，MySQL Server 层根据 SQL 语句的语义确定的锁，通过 `ha_innabase::store_lock()` 函数把锁传入 InnoDB 层，这体现在：根据 SQL 语句和隔离级别给“`m_prebuilt->select_lock_type`”和“`m_prebuilt->stored_select_lock_type`”赋予不同的值。

```
THR_LOCK_DATA**
ha_innabase::store_lock(
    THD*          thd, /*!< in: user thread handle */
    THR_LOCK_DATA** to, /*!< in: pointer to the current element in an array
                        of pointers to lock structs; only used as return value */
    thr_lock_type  lock_type) /*!< in: lock type to store in 'lock'; this may
                        also be TL_IGNORE */
{
    ...
    if (srv_read_only_mode //如果是只读模式
        && !dict_table_is_intrinsic(m_prebuilt->table) //且不是系统表即是用户表
        && (sql_command == SQLCOM_UPDATE //则遇到如下命令时，报告警告信息
            || sql_command == SQLCOM_INSERT
            || sql_command == SQLCOM_REPLACE
            || sql_command == SQLCOM_DROP_TABLE
            || sql_command == SQLCOM_ALTER_TABLE
            || sql_command == SQLCOM_OPTIMIZE
            || (sql_command == SQLCOM_CREATE_TABLE
                //类似CREATE TABLE...SELECT...FROM UPDATE，锁不属于读锁类型而是写锁类型
                则要报告警告
                && (lock_type >= TL_WRITE_CONCURRENT_INSERT && lock_type <= TL_WRITE)))
        || sql_command == SQLCOM_CREATE_INDEX
        || sql_command == SQLCOM_DROP_INDEX
        || sql_command == SQLCOM_DELETE)) {
        ib_senderrf(trx->mysql_thd, IB_LOG_LEVEL_WARN, ER_READ_ONLY_MODE);
    } else if (sql_command == SQLCOM_FLUSH && lock_type == TL_READ_NO_INSERT) { /*
        Check for FLUSH TABLES ... WITH READ LOCK */
        ...
        if (trx->isolation_level == TRX_ISO_SERIALIZABLE) {
            //对于FLUSH TABLES命令，根据隔离级别施加锁
            m_prebuilt->select_lock_type = LOCK_S;
```





```

        //序列化隔离级别施加读锁，这是实现序列化的关键，用共享锁排斥其他写锁
        m_prebuilt->stored_select_lock_type = LOCK_S;
    } else {
        m_prebuilt->select_lock_type = LOCK_NONE;
        m_prebuilt->stored_select_lock_type = LOCK_NONE;
    }
} else if (sql_command == SQLCOM_DROP_TABLE) {
    //如上一节的例1所示，删除表是在表的元数据上施加了排它锁，故不对锁做调整
    /* MySQL calls this function in DROP TABLE though this tablehandle may
    belong to another thd that is running a query.
    Let us in that case skip any changes to the m_prebuilt struct. */

    } else if ((lock_type == TL_READ && in_lock_tables)/* Check for LOCK TABLE
    t1,...,tn WITH SHARED LOCKS */
        || (lock_type == TL_READ_HIGH_PRIORITY && in_lock_tables)
        || lock_type == TL_READ_WITH_SHARED_LOCKS
        || lock_type == TL_READ_NO_INSERT
        || (lock_type != TL_IGNORE && sql_command != SQLCOM_SELECT)) {
        /* The OR cases above are in this order:
        1) MySQL is doing LOCK TABLES ... READ LOCAL, or we are processing a
        stored procedure or function, or
        2) (we do not know when TL_READ_HIGH_PRIORITY is used), or
        3) this is a SELECT ... IN SHARE MODE, or
        4) we are doing a complex SQL statement like INSERT INTO ... SELECT ...
        and the logical logging (MySQL binlog)
        requires the use of a locking read, or MySQL is doing LOCK TABLES ... READ.
        5) we let InnoDB do locking reads for all SQL statements that are not
        simple SELECTs;
        note that select_lock_type in this case may get strengthened in
        ::external_lock() to LOCK_X.
        Note that we MUST use a locking read in all data modifying SQL statements,
        because otherwise the execution would not be serializable, and also
        the results from the update could be
        unexpected if an obsolete consistent read view would be used. */
        /* Use consistent read for checksum table */
        if (sql_command == SQLCOM_CHECKSUM //CHECKSUM TABLE只施加的读锁在Parser阶
        段完成，要使用一致性读，所以 "select_lock_type = LOCK_NONE"
            || ((srv_locks_unsafe_for_binlog || trx->isolation_level <= TRX_ISO_READ_COMMITTED)
                && trx->isolation_level != TRX_ISO_SERIALIZABLE
                && (lock_type == TL_READ || lock_type == TL_READ_NO_INSERT)
                && (sql_command == SQLCOM_INSERT_SELECT || sql_command == SQLCOM_
                REPLACE_SELECT
                    || sql_command == SQLCOM_UPDATE || sql_command == SQLCOM_
                    CREATE_TABLE))) {
            /* If we either have innobase_locks_unsafe_for_binlog option set or
            this session is using READ COMMITTED

```



```

        isolation level and isolation level of the transaction is not set
        to serializable and MySQL is doing
        INSERT INTO...SELECT or REPLACE INTO...SELECT or UPDATE ... =
        (SELECT ...) or
        CREATE ...SELECT... without FOR UPDATE or IN SHAREMODE in select,
        then we use consistent read for select. */
        m_prebuilt->select_lock_type = LOCK_NONE;
        m_prebuilt->stored_select_lock_type = LOCK_NONE;
    } else {
        m_prebuilt->select_lock_type = LOCK_S; //上述情况之外的所有其他情况, 使用读锁
        m_prebuilt->stored_select_lock_type = LOCK_S;
    }
} else if (lock_type != TL_IGNORE) {
    /* We set possible LOCK_X value in external_lock, not yet here even if
    this would be SELECT ... FOR UPDATE */
    m_prebuilt->select_lock_type = LOCK_NONE; //后续一些情况交由external_lock()处理
    m_prebuilt->stored_select_lock_type = LOCK_NONE;
}
...
}

```

如下是“LOCK TABLES...”命令调用 ha\_innobase::store\_lock() 函数执行栈信息：

```

ha_innobase::store_lock(THD * thd, st_thr_lock_data * to, thr_lock_type lock_
type) Line 15917C++ //进入InnoDB层
get_lock_data(THD * thd, TABLE * * table_ptr, unsigned int count, unsigned int
flags) Line 728C++//属于MySQL Server层
mysql_lock_tables(THD * thd, TABLE * * tables, unsigned int count, unsigned int
flags) Line 322C++
lock_tables(THD * thd, TABLE_LIST * tables, unsigned int count, unsigned int
flags) Line 6398C++ //执行LOCK TABLES, 属于MySQL Server层。许多表级的封锁操作, 都通过
lock_tables()函数检查封锁情况、进行加锁操作
lock_tables_open_and_lock_tables(THD * thd, TABLE_LIST * tables) Line 2092C++
mysql_execute_command(THD * thd, bool first_level) Line 3528C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command
command) Line 1251C++
do_command(THD * thd) Line 819C++

```

MySQL Server层根据 SQL 语句的语义确定的锁，通过 ha\_innobase::external\_lock() 函数根据 SQL 语句和隔离级别对锁进行调整。

```

int
ha_innobase::external_lock(
    THD*   thd,          /*!< in: handle to the user thread */
    int    lock_type)    /*!< in: lock type */
{...

```





```

if (lock_type == F_WRLCK) {
    /* If this is a SELECT, then it is in UPDATE TABLE ...or SELECT ... FOR UPDATE */
    m_prebuilt->select_lock_type = LOCK_X; // UPDATE TABLE ...或SELECT ...
    FOR UPDATE施加排它锁
    m_prebuilt->stored_select_lock_type = LOCK_X;
}

if (lock_type != F_UNLCK) {...
    if (trx->isolation_level == TRX_ISO_SERIALIZABLE //隔离级别是可串行化
        && m_prebuilt->select_lock_type == LOCK_NONE
        && thd_test_options(thd, OPTION_NOT_AUTOCOMMIT | OPTION_BEGIN)) {
        //自动提交
        /* To get serializable execution, we let InnoDBconceptually add 'LOCK
           IN SHARE MODE' to all SELECTs
           which otherwise would have been consistent reads.
           An exception is consistent reads in the AUTOCOMMIT=1 mode:
           we know that they are read-only transactions, and theycan be
           serialized also if performed as consistentreads. */
        //当自动提交且隔离级别是可串行化的情况下，“概念上”加读锁，实际是物理加读锁，这样
        才能保证可串行化的效果，因为读锁排斥写锁
        m_prebuilt->select_lock_type = LOCK_S; //注意此处是实现可串行化的一段重要代码
        m_prebuilt->stored_select_lock_type = LOCK_S;
    }
}
...
} else {
    TrxInInnoDB::end_stmt(trx);
    DEBUG_SYNC_C("ha_innobase_end_statement");
}
...
}

```

如下是“LOCK TABLES...”命令调用 ha\_innobase::external\_lock() 函数执行栈信息：

```

ha_innobase::external_lock(THD * thd, int lock_type) Line 14895C++ //进入InnoDB层
handler::ha_external_lock(THD * thd, int lock_type) Line 7669C++//属于MySQL Server层
lock_external(THD * thd, TABLE ** tables, unsigned int count) Line 391C++
mysql_lock_tables(THD * thd, TABLE ** tables, unsigned int count, unsigned int
flags) Line 328C++
lock_tables(THD * thd, TABLE_LIST * tables, unsigned int count, unsigned int
flags) Line 6398C++//执行LOCK TABLES, 属于MySQL Server层
lock_tables_open_and_lock_tables(THD * thd, TABLE_LIST * tables) Line 2092C++
mysql_execute_command(THD * thd, bool first_level) Line 3528C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command
command) Line 1251C++
do_command(THD * thd) Line 819C++

```



如下再通过一条“SELECT \* FROM bluesea WHERE c1=2;”命令来看 ha\_innobase::store\_lock() 函数执行栈信息：

```
ha_innobase::store_lock(THD * thd, st_thr_lock_data * * to, thr_lock_type lock_type) Line 15841 C++ //进入InnoDB层
get_lock_data(THD * thd, TABLE * * table_ptr, unsigned int count, unsigned int flags) Line 728 C++ //属于MySQL Server层
mysql_lock_tables(THD * thd, TABLE * * tables, unsigned int count, unsigned int flags) Line 322 C++
lock_tables(THD * thd, TABLE_LIST * tables, unsigned int count, unsigned int flags) Line 6398 C++ //执行LOCK TABLES, 属于MySQL Server层
handle_query(THD * thd, LEX * lex, Query_result * result, unsigned __int64 added_options, unsigned __int64 removed_options) Line 157 C++
execute_sqlcom_select(THD * thd, TABLE_LIST * all_tables) Line 4899 C++
mysql_execute_command(THD * thd, bool first_level) Line 2564 C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305 C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command command) Line 1251 C++
do_command(THD * thd) Line 819 C++
```

对于普通的 SELECT 语句，在 ha\_innobase::store\_lock() 中为锁赋值“m\_prebuilt->select\_lock\_type = LOCK\_NONE”，这是因为普通的 SELECT 语句采取的是一致性读的方式，第12章将对基于 MVCC 技术的一致性读详细讨论。

而这样普通的 SELECT 语句，同样会进入 ha\_innobase::external\_lock() 函数，进行与锁相关的进一步判断（判断方式，请关注 ha\_innobase::external\_lock() 函数的代码分析中的特别说明）。

```
ha_innobase::external_lock(THD * thd, int lock_type) Line 15138C++ //进入InnoDB层
handler::ha_external_lock(THD * thd, int lock_type) Line 7669C++
//属于MySQL Server层
lock_external(THD * thd, TABLE * * tables, unsigned int count) Line 391C++
mysql_lock_tables(THD * thd, TABLE * * tables, unsigned int count, unsigned int flags) Line 328C++
lock_tables(THD * thd, TABLE_LIST * tables, unsigned int count, unsigned int flags) Line 6398C++
handle_query(THD * thd, LEX * lex, Query_result * result, unsigned __int64 added_options, unsigned __int64 removed_options) Line 157C++
execute_sqlcom_select(THD * thd, TABLE_LIST * all_tables) Line 4899C++
mysql_execute_command(THD * thd, bool first_level) Line 2564C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command command) Line 1251C++
do_command(THD * thd) Line 819C++
```

另外，通常情况下，当一条 SQL 语句执行结束后，还会进入到 ha\_innobase::store\_





lock() 函数进行锁的处理, 如下是 “SELECT \* FROM bluesea WHERE c1=2;” 语句执行结束时刻的栈信息。

```
ha_innbase::store_lock(THD * thd, st_thr_lock_data * * to, thr_lock_type lock_
type) Line 15841 C++
get_lock_data(THD * thd, TABLE * * table_ptr, unsigned int count, unsigned int
flags) Line 728 C++
mysql_unlock_some_tables(THD * thd, TABLE * * table, unsigned int count)
Line 432C++ //解锁, 但实际的解锁操作是调用mysql_unlock_tables()来完成的, 这样会又进入
ha_innbase::external_lock()函数, 重置一些变量的值
JOIN::optimize() Line 411 C++
SELECT_LEX::optimize(THD * thd) Line 1024 C++
handle_query(THD * thd, LEX * lex, Query_result * result, unsigned __int64 added_
options, unsigned __int64 removed_options) Line 170 C++
execute_sqlcom_select(THD * thd, TABLE_LIST * all_tables) Line 4899 C++
mysql_execute_command(THD * thd, bool first_level) Line 2564 C++
mysql_parse(THD * thd, Parser_state * parser_state) Line 5305 C++
dispatch_command(THD * thd, const COM_DATA * com_data, enum_server_command
command) Line 1251 C++
do_command(THD * thd) Line 819 C++
```

#### 4. 锁的施加原则

在 11.5.3 节, 首先讨论了加锁的粒度起源于 SQL 语句的语义, 然后重点讨论了从到 InnoDB 层的锁的语义的传递过程。但是, 没有讨论什么情况下应当施加什么锁?

这个问题, 对于基于锁的并发访问控制技术的实现而言, 是一个重要的话题。

其实, 在 5.2.3 节, 我们就讨论过 InnoDB 的锁的施加原则, 5.2.3 节对于各种需要加锁的 SQL 语句进行了加锁情况的说明, 详情请参见 5.2.3 节, 但这些规则不仅是针对 InnoDB 的, 而是在 MySQL Server 层这一级即确定的。

另外, 在 11.3.1 节的 “5 显式锁与隐式锁”, 对锁在不同 SQL 语句执行时锁的施加情况做了一些探讨。在 11.4.1 节的 “3 元数据锁的粒度”, 也对各种锁的施加情况做了详细的分析, 请反复阅读 11.4.1 节。

所以, 作为一名数据库内核开发的工程师, 在实现封锁并发控制机制的时候, 需要全盘考虑系统锁和事务锁, 在事务锁范围内考虑元数据锁和元组级锁的实现策略和施加规则, 并考虑相关的死锁检测机制和释放锁的时机等内容。

#### 5. 元数据锁与记录锁的关系

元数据锁, 控制的是 DDL 的并发; 记录锁, 控制的是 DQL、DML 之间的并发。二者相似之处在于都是事务类别的锁, 不同的是控制的操作不同、作用的对象不同。对于表这个特殊对象, 元数据锁作用在表的结构上, 记录作用在数据项 (索引的记录) 上。

#### 6. 其他

除了前面提及的一些锁相关内容外, 还有其他一些函数, 也与锁相关, 比如 lock\_



tables\_check() 函数在语义层面对锁做合法性检查，以及其上层的调用函数 open\_ltable()、lock\_tables() 等函数（如图 11-10 所示）、解锁相关的 mysql\_unlock\_tables() 函数和 lock.cc 文件中的其他一些函数如与临时表相关的锁操作、如 MyISAM 等其他引擎等，都与锁操作密切相关，在做工程实现的时候，都需要仔细考虑才能做出正确的设计。这些内容较多，限于篇幅，本书不再逐一深入讨论，请自行参看源码。

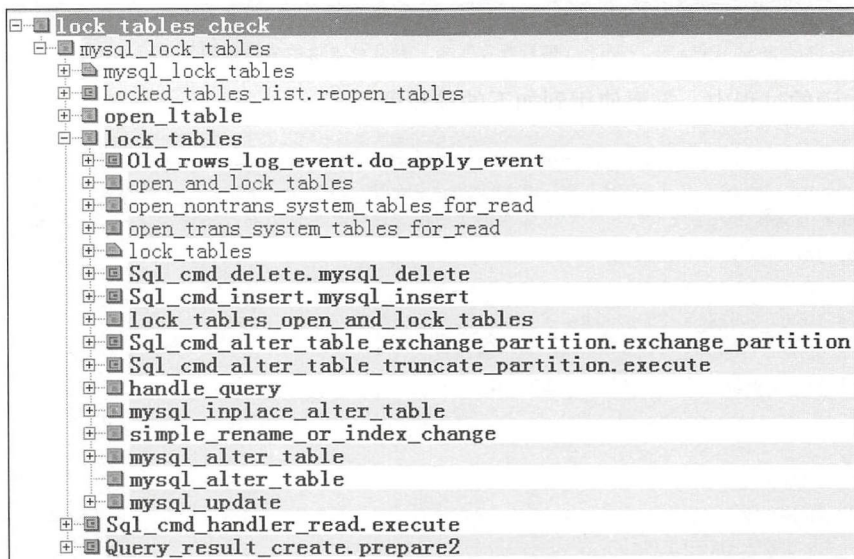


图 11-10 lock\_tables\_check() 函数调用关系图

## 11.6 其他类型的锁

MySQL 中还存在其他类型的锁，下面将介绍一种系统锁和一种事务锁。

### 11.6.1 Mini-Transaction 加锁

在 10.3.6 和 10.3.7 节，介绍了 Mini-Transaction。在本节，将讨论 Mini-Transaction 相关的系统锁操作，尽管 Mini-Transaction 独立使用了四个加锁操作，但从本质上看，还是属于系统锁类型。

```

/** Lock an rw-lock in s-mode. */
#define mtr_s_lock(l, m) (m)->s_lock((l), __FILE__, __LINE__)
//施加读锁，通过调用rw_lock_s_lock_func()函数实现施加读锁
/** Lock an rw-lock in x-mode. */
#define mtr_x_lock(l, m) (m)->x_lock((l), __FILE__, __LINE__) //施加写锁
/** Lock a tablespace in x-mode. */

```





```

#define mtr_x_lock_space(s, m) (m)->x_lock_space((s), __FILE__, __LINE__)
//在指定表空间上施加读锁, 通过mtr_t::x_lock_space()完成⊖
/** Lock an rw-lock in sx-mode. */
#define mtr_sx_lock(l, m) (m)->sx_lock((l), __FILE__, __LINE__) //施加读写锁

#define rw_lock_s_unlock(L) rw_lock_s_unlock_gen(L, 0)
//调用rw_lock_s_unlock_func()函数释放读锁, 释放被上面的宏施加的读锁
#define rw_lock_x_unlock(L) rw_lock_x_unlock_gen(L, 0)
//调用rw_lock_x_unlock_func()函数释放写锁, 释放被上面的宏施加的写锁

```

在施加锁的过程中, 需要使用到如下的锁粒度。

```

/** Types for the mlock objects to store in the mtr memo; NOTE that the first 3
values must be RW_S_LATCH, RW_X_LATCH, RW_NO_LATCH */
enum mtr_memo_type_t {           //各个粒度的锁可以顾名思义, 所以不再多述
#ifdef UNIV_INNOCHECKSUM
    MTR_MEMO_PAGE_S_FIX = RW_S_LATCH,
    MTR_MEMO_PAGE_X_FIX = RW_X_LATCH,
    MTR_MEMO_PAGE_SX_FIX = RW_SX_LATCH,
    MTR_MEMO_BUF_FIX = RW_NO_LATCH,
#endif /* !UNIV_CHECKSUM */

#ifdef UNIV_DEBUG
    MTR_MEMO_MODIFY = 32,
#endif /* UNIV_DEBUG */

    MTR_MEMO_S_LOCK = 64,
    MTR_MEMO_X_LOCK = 128,
    MTR_MEMO_SX_LOCK = 256
};

```

上述讨论的是 Mini-Transaction 锁的施加。有施加就有释放, Mini-Transaction 锁的释放, 是通过 `mtr_t::memo_release()` 来完成的。

而 Mini-Transaction 锁针对的对象, 是一个 block 或称为页面的物理数据块 (用 “`struct buf_block_t`” 定义), 即磁盘式数据库以物理页面为单位对数据进行页面管理 (页是数据缓冲区中的基本操作单位, 从中解析页头、元组等; 也是数据从存储层加载的基本单位)。

## 11.6.2 事务锁之谓词锁

在 11.3.2 节, 讨论了间隙类的锁 (GAP 或 next key locking)。InnoDB 的表结构的组织方式是索引组织表 (Index Organize Table, 简称 IOT), 记录锁的本质是在索引上面施加锁, 而这样的索引本身是有序的, 所以在有序的索引上通过在一定的范围上加锁就能够锁定一个范围使得这个范围内的数据不发生变化, 因而能够避免幻象异常现象。

<sup>⊖</sup> 可参阅 `rw_lock_x_lock_func()` -> `rw_lock_x_lock_low()` 等函数查看实现细节。



但是,对于空间索引 (GIS Index),其不再是单方向有序,而是多维度的数据模型,所以不能有单向有序的特性可以被利用,因此使用间隙类的锁是避免不了幻象异常现象的。InnoDB 为了在空间索引上支持 “repeated read” 和 “serializable read” 这两个隔离级别,引入了谓词锁 (Predicate lock) 的概念来解决幻象现象。

InnoDB 在空间索引上的谓词锁,是锁定一个 MBR(minimum bounding rectangle/box)<sup>①</sup>,一旦一个 MBR 被锁定,其他会话不能够在这个范围内进行插入或更新操作。这时,就意味着并发情况下、使得在 “repeated read” 和 “serializable read” 这两个隔离级别下,可以避免幻象现象<sup>②</sup>。

原理清晰之后,接下来讨论 InnoDB 是怎么在空间索引上实现谓词锁的<sup>③</sup>,InnoDB 对于谓词锁的代码位于 lock0prdt.cc 和 lock0prdt.h 文件中。

## 1. 谓词锁的施加

首先,定义了表示谓词锁的数据结构:

```
typedef struct lock_prdt {
    void*      data; /* Predicate data */
    uint16     op;   /* Predicate operator */
} lock_prdt_t;
```

其次,通过 lock\_prdt\_lock() 函数完成加锁。这时,就需要注意,在什么对象上加锁,锁是怎么在内存中保存的。

```
/******//**
Acquire a predicate lock on a block
@return DB_SUCCESS, DB_LOCK_WAIT, DB_DEADLOCK, or DB_QUEUE_THR_SUSPENDED */
//同样存在死锁的可能
dberr_t
lock_prdt_lock(
    buf_block_t*    block, /*!< in/out: buffer block of rec */
    //被加锁的对象,是一个数据块,但加锁信息记录在块头,所以使用了控制块
    lock_prdt_t*    prdt, /*!< in: Predicate for the lock */ //谓词锁
    dict_index_t*    index, /*!< in: secondary index */
    lock_mode        mode, /*!< in: mode of the lock which the read cursor
should set on records: LOCK_S or LOCK_X;
                                the latter is possible in SELECT FOR UPDATE */
    uint            type_mode, /*!< in: LOCK_PREDICATE or LOCK_PRDT_PAGE */
    //LOCK_PREDICATE是谓词锁,LOCK_PRDT_PAGE是页锁
    que_thr_t*      thr, /*!< in: query thread (can be NULL if BTR_NO_
LOCKING_FLAG) */
```

① MBR, 最小外包矩形。就是包围图元,且平行于X, Y轴的最小外接矩形,是GIS (Geographic Information System) 或者计算机图形学上非常重要的概念。

② 这也说明了谓词锁和隔离级别的关系。

③ 参考: <http://dev.mysql.com/worklog/task/?spm=5176.100239.blogcont4270.8.Nn4D01&id=6609>





```

mtr_t*      mtr)      /*!< in/out: mini-transaction */
{...
    if (trx->read_only || dict_table_is_temporary(index->table)) {
        //只读事务或使用临时表（临时表属于特定会话无并发）则不加锁
        return(DB_SUCCESS);
    }
    ...
    hash_table_t* hash = type_mode == LOCK_PREDICATE
        //根据锁的mode，获取在全局锁表中的对应的hash锁表
        ? lock_sys->prdt_hash
        : lock_sys->prdt_page_hash;
    ...
    const ulint prdt_mode = mode | type_mode;
    //形成要申请的锁的最终标识，如mode为LOCK_S，type_mode为LOCK_PREDICATE
    lock_t* lock = lock_rec_get_first_on_page(hash, block);
    //在要加锁的对象上看，是否有旧的锁存在，便于并发冲突判断
    //锁是存在于被加锁对象上的，也会被注册到对应的hash表上，如lock_sys的prdt_hash 或prdt_page_hash

    if (lock == NULL) { //在要加锁的对象上看到没有锁存在，意味着申请加锁成功
        RecLock rec_lock(index, block, PRDT_HEAPNO, prdt_mode);
        //则生成一个新的记录锁在指定的索引index上
        lock = rec_lock.create(trx, false, true);
        //第三个参数值true表示要加到hash表上（prdt_hash或prdt_page_hash）
        status = LOCK_REC SUCCESS_CREATED;
    } else { //在要加锁的对象上看到有锁存在，可能加锁会不成功
        trx_mutex_enter(trx);
        if (lock_rec_get_next_on_page(lock)
            //根据锁能找到与之匹配的合适的记录锁在页面上
            || lock->trx != trx //不是同一个会话
            || lock->type_mode != (LOCK_REC | prdt_mode) //不是记录锁与谓词锁
            || lock_rec_get_n_bits(lock) == 0 //锁标志位没有置位
            || ((type_mode & LOCK_PREDICATE) && (!lock_prdt_consistent(lock_get_
                prdt_from_lock(lock), prdt, 0))))
            //是谓词锁，但已经存在的锁和新申请的锁不相容，相容性通过
            //lock_prdt_consistent()判断⊖
        {
            lock = lock_prdt_has_lock(mode, type_mode, block, prdt, trx);
            //在指定页面，检查是否有已经授予的更强的或相当的锁
            if (lock == NULL) { //在指定页面，不存在“有已经授予的更强的或相当的锁”
                const lock_t* wait_for;
                wait_for = lock_prdt_other_has_conflicting(prdt_mode, block, prdt, trx);
                //检查是否有其他会话持有这样的锁
                if (wait_for != NULL) {
                    //检查是否有其他会话持有这样的锁，如有则本会话需要等待

```

⊖ 空间索引的谓词锁的相容性判断，是根据对空间索引的操作进行一个矩形范围判断。参见lock\_prdt\_consistent()。



```

        RecLock rec_lock(thr, index, block, PRDT_HEAPNO, prdt_mode, prdt);
        err = rec_lock.add_to_waitq(wait_for);
        //需要等待则加入等待队列，会进行死锁判断（参见11.3.2节第5小节）
    } else {
        //检查是否有其他会话持有这样的锁，如没有则本会话可以加锁
        lock_prdt_add_to_queue(prdt_mode, block, index, trx, prdt, true);
        //把锁加入谓词锁队列，注册到hash表中
        status = LOCK_REC_SUCCESS;
    }
}

trx_mutex_exit(trx);
} else { //同一个会话内（即同一个事务）、不能根据锁能找到与之匹配的合适的记录锁在页面上等if没有限定的情况都走到这里
    trx_mutex_exit(trx);
    if (!lock_rec_get_nth_bit(lock, PRDT_HEAPNO)) {
        lock_rec_set_nth_bit(lock, PRDT_HEAPNO);
        status = LOCK_REC_SUCCESS_CREATED;
    }
}

lock_mutex_exit();

if (status == LOCK_REC_SUCCESS_CREATED && type_mode == LOCK_PREDICATE) {
    //加谓词锁成功
    /* Append the predicate in the lock record */
    lock_prdt_set_prdt(lock, prdt);
    //加谓词锁成功，则把谓词锁保存（通过内存拷贝）在lock对象上
}

return(err);
}

```

## 2. 谓词锁施加的条件

上述讨论了谓词锁的加锁过程，那么，什么时候需要施加谓词锁呢？下面以空间索引搜索为例，来讨论谓词锁施加的条件（谓词锁的释放，参见 11.3.2 的第 7 小节）。

```

/** Searches an index tree and positions a tree cursor on a given level. ... */
void
btr_cur_search_to_nth_level(...) //搜索索引树
{...
    /* Add Predicate lock if it is serializable isolation and only if it
       is in the search case */
    if (dict_index_is_spatial(index) //如果是空间索引
        && cursor->rtr_info->need_prdt_lock
        //如果需要谓词锁，参见图11-10（根据隔离级别确定是否需要谓词锁）
        && mode != PAGE_CUR_RTREE_INSERT
        && mode != PAGE_CUR_RTREE_LOCATE
        && mode >= PAGE_CUR_CONTAIN) {

```



```

...
        lock_prdt_lock(block, &prdt, index, LOCK_S, LOCK_PREDICATE, cursor->thr, mtr);
...
    }
...
}

```

“need\_prdt\_lock”变量被多个函数赋值(如图11-11所示),其中只有row\_search\_mvcc()函数赋值为true(其他函数对need\_prdt\_lock的赋值可自行查阅如图11-10中指出的相关代码),表示在做索引读(此处对应的是空间索引)的时候需要施加谓词锁。具体代码如下:

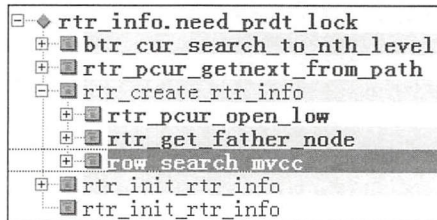


图 11-11 “need\_prdt\_lock”被row\_search\_mvcc()赋值为true

```

row_search_mvcc(...) //分为几个阶段,其中,对空间索引是否需要谓词锁在“PHASE 3”进行判断
{
...
    /* PHASE 0: Release a possible s-latch we are holding on the
       adaptive hash index latch if there is someone waiting behind */
...
    /* PHASE 1: Try to pop the row from the prefetch cache */
...
    /* PHASE 2: Try fast adaptive hash index search if possible */
...
    /* PHASE 3: Open or restore index cursor position */
...
    /* Open or restore index cursor position */
    if (UNIV_LIKELY(direction != 0)) {
...
    } else if (dtuple_get_n_fields(search_tuple) > 0) {
...
        if (dict_index_is_spatial(index)) { //如果是空间索引
            bool    need_pred_lock;
            need_pred_lock = (set_also_gap_locks
                //需要加GAP间隙锁且隔离级别大于读已提交,则为need_pred_lock赋值
                && !(srv_locks_unsafe_for_binlog
                    || trx->isolation_level <= TRX_ISO_READ_COMMITTED)
                //隔离级别大于读已提交才能使用谓词锁
                && prebuilt->select_lock_type != LOCK_NONE);
            ...
        }
...
}

```

## 11.7 事务与锁

事务与锁之间的关联点在什么地方呢？

首先，锁发生在一个事务内部，所以锁信息依附于一个事务。一个会话上，同一时刻只能有一个事务，因此在一个会话上事务是一个会话级的全局信息。可以认为锁就是会话级别上的信息。

```

/** The locks and state of an active transaction. Protected by lock_sys->mutex,
    trx->mutex or both. */
struct trx_lock_t { //某一个活跃事务的锁及其状态
    uint      n_active_thrs;    /*!< number of active query threads */
    trx_que_t  que_state;       /*!< valid when trx->state
                                包含有四种可顾名思义的状态: TRX_QUE_RUNNING, TRX_QUE_LOCK_WAIT,
                                TRX_QUE_ROLLING_BACK, TRX_QUE_COMMITTING */
    lock_t*    wait_lock;       /*指向请求的锁
    ib_uint64_t deadlock_mark;  /*死锁标志
    bool       was_chosen_as_deadlock_victim; //是否被选为了受害者
    time_t     wait_started;    /*处于锁等待的起始时间，便于用锁超时进行时间检测判断
    que_thr_t*  wait_thr;       /*正在等哪个会话

    lock_pool_t  rec_pool;      /*!< Pre-allocated record locks */
    //一个事务上预先分配的记录锁缓存池。lock_pool_t的定义⊖
    lock_pool_t  table_pool;    /*!< Pre-allocated table locks */
    //一个事务上预先分配的表锁缓存池
    uint        rec_cached;     /*!< Next free rec lock in pool */

    uint        table_cached;   /*!< Next free table lock in pool */
    mem_heap_t*  lock_heap⊖;  /*锁的内存空间，是从这个堆上申请的
    trx_lock_list_t  trx_locks; //locks requested by the transaction;
    //事务已经申请到的事务的锁的双向列表

    lock_pool_t  table_locks;   //本事务内的所有的表锁
    bool         cancel;        //本事务是否被取消（被回滚或超时等待发生将认为事务被取消）
    uint         n_rec_locks;    //本事务内的所有记录锁

    /** The transaction called ha_innobase::start_stmt() to lock a table. Most
        likely a temporary table. */
    bool         start_stmt;
};

```

其次，因为并发操作的存在，死锁将发生，如果只把锁和会话紧密绑定，则不能探测

⊖ lock\_pool\_t的定义：typedef std::vector<ib\_lock\_t\*, ut\_allocator<ib\_lock\_t\*>>lock\_pool\_t;

⊖ 在TrxFactory的init()方法中：trx->lock.lock\_heap = mem\_heap\_create\_typed(1024, MEM\_HEAP\_FOR\_LOCK\_HEAP);



死锁。所以，需要把所有的行级锁信息，注册到整个数据库引擎实例这一层，这样才能检测死锁。所以才有如下的全局级别的“lock\_sys\_t\*lock\_sys”行级锁表。

```

/** The lock system struct */
struct lock_sys_t{    //全局的行级锁表结构，所有的行级锁都在这个结构体定义的lock_sys中注册
    char                pad1[CACHE_LINE_SIZE];
    LockMutex           mutex;
    hash_table_t*       rec_hash;    //全局的记录锁Hash表，所有的记录锁注册到这个Hash表里
    hash_table_t*       prdt_hash;   //全局的谓词锁Hash表，所有的谓词锁注册到这个Hash表里
    hash_table_t*       prdt_page_hash; //全局的谓词页锁Hash表，所有的谓词页锁注册到这个Hash表里
    char                pad2[CACHE_LINE_SIZE];    /*!< Padding */
    LockMutex           wait_mutex;    //保护下面几个成员的系统锁
    srv_slot_t*         waiting_threads; //正在等待的线程会话有哪些
    srv_slot_t*         last_slot;     //在waiting_threads中的最高（最后）的一个槽
    ibool               rollback_complete;
    //所有的回滚的事务完成（伴随着事务上的锁要被释放，如调用lock_rec_discard()）
    ulint               n_lock_max_wait_time;    /*!< Max wait time */
    os_event_t          timeout_event;           /*!< Set to the event that is created
in the lock wait monitor thread.
    bool               timeout_thread_active;    /*!< True if the timeout thread is running */
};

```

在 InnoDB 引擎内存，查找锁（lock\_rec\_has\_expl() 等），将通过 lock\_sys 全局变量进行。所以 lock\_sys 使用之处十分广泛，可自行查阅代码，本文不再赘述。

全局行级锁表的初始化，是在 InnoDB 的初始化的时候完成的，全局锁表初始化函数 lock\_sys\_create()。调用关系如下：

```

innobase_init()
-> innobase_start_or_create_for_mysql()
-> lock_sys_create()

```

## 11.8 本章小结

本章辨析了 MySQL Server 层和 InnoDB 层与事务锁相关的内容，从而帮助读者全面掌握 MySQL 基于封锁的并发访问控制技术。

在这一章，讨论了两种锁，一是系统锁，二是事务锁，其中，11.2 节的系统锁是控制共享对象不被并发修改破坏的，也是事务锁实现的物理基础之一；而事务锁是重点，主要讨论了两层事务锁，一是 11.3 节讨论的控制 DML 语句并发的记录锁，二是 11.4 节讨论的控制 DDL 语句并发的元数据锁，即通常所说的行级锁。

如 11.5 节分析，锁的施加规则，首先取决于 SQL 语句的语义，而 MySQL Server 层的分析器（Parser）在做词法语法分析的时候即完成加锁语义的解析，所以加锁的过程是

从 MySQL Server 层到 InnoDB 层的一个逐步传递的过程（详见 11.5 节），只是这个过程中元数据锁的主要在 MySQL Server 层完成，行级锁（11.3.2、11.3.3 节讨论）虽然在 MySQL Server 层定义并向下传递，但是与记录锁锁相关的主体代码位于 InnoDB 层。

最后几节，讨论了与锁相关的其他一些重要内容，因篇幅所限，还有一部分锁相关的内容没有涉及，请读者自行阅读代码，以全面掌握封锁技术的各种细节。



## 第 12 章

# InnoDB 并发控制系统的实现——MVCC

MVCC，多版本并发访问控制技术。本书在 2.2.4 和 2.2.5 节做过理论上的介绍。本章我们也把 MVCC 技术单独列为一章，这是因为 MVCC 对于传统的数据库引擎而言，非常重要。但是，为什么 MVCC 非常重要呢？

大家都知道 MVCC 这个词很火，很多开发数据库引擎的团队宣传其数据库产品的时候，把 MVCC 作为一个重点功能做了宣传，似乎数据库的并发控制技术使用的只是 MVCC 技术。宣传书中并不会说明 MVCC 技术和其他并发访问控制技术之间的关系。那么，MVCC 技术和其他并发访问控制技术之间的关系到底是什么？

细心的读者，在阅读了第 2 章之后，其实完全可以回答这两个问题。

在此对这两个问题再做一个小结：

Q1：MVCC 技术和其他并发访问控制技术之间的关系到底是什么？

答：

- 首先，基于封锁技术、基于时间戳技术、基于有效性检查、MVCC 等技术，都是并发访问控制技术。
- 其次，多数数据库引擎，采用的并发访问控制技术，都是基于封锁技术，然后在封锁技术的基础上，采用了 MVCC 技术。如 Oracle、PostgreSQL、MySQL/InnoDB 都是如此。
- 第三，MVCC 技术多用于弥补基于封锁技术在读 - 写、写 - 读这两种情况下的并发不足（不足之处是根本没有并发，MVCC 的引入使得并发可行）。

Q2：为什么 MVCC 十分重要呢？

答：

- 在第 1 章和第 2 章，讨论了四种情况（读 - 读、读 - 写、写 - 读、写 - 写）里的三种

冲突（读-写、写-读、写-写），基于封锁的并发访问控制技术，为了保证数据的一致性，只能允许读-读操作并发执行，而其他三种情况会造成如第1章里讨论的一些数据异常现象，所以被限制了并发，这样导致基于封锁的并发访问控制技术的并发度很低。

□ MVCC 技术，因为同一个数据项存在多个版本，使得不同事务的对同一个数据项的读操作可以根据其读时刻的快照作用在不同的版本上，因而避免了 1.1 节讨论的三种读数据异常现象，所以对于读-写、写-读操作允许并发，提高了并发度。所以 MVCC 技术才显得重要。

□ 传统的事务型数据库，尽管以插入和修改数据的操作为主，但是查询工作也很重要，所以 MVCC 技术引入使得读和写操作互不阻塞、使得长的读操作事务能够被放心地并发执行，大大提高了事务型数据库在分析、查询方面的能力，因而被传统数据库厂商所青睐。

□ 基于上面所述的原因，才使得 MVCC 技术显得十分重要。

尽管本章把 MVCC 技术单独成章，从逻辑的角度看，在理解的时候，不要把 MVCC 和其他并发访问控制技术割裂，而是要把他们融合在一起加以理解。

InnoDB 的 MVCC 技术解决了其使用的基于封锁技术在并发方面的不足之处（不足之处如 Q2 所言），如下我们讲讨论 InnoDB 对 MVCC 技术的实现方式。

## 12.1 数据结构

### 12.1.1 MVCC

MVCC 类，是一个管理者，管理“read view”对象，即管理快照。作为一个管理者，就对应创建、关闭等管理功能。

```
/** The MVCC read view manager */
class MVCC { //MVCC类，是一个管理者，管理“read view”对象，即管理快照
...
    /** Allocate and create a view.
    @param view    view owned by this class created for the caller. Must be freed
    by calling close()
    @param trx     transaction creating the view */
    void view_open(ReadView*& view, trx_t* trx); //创建一个read view

    /** Close a view created by the above function.
    @para view     view allocated by trx_open.
    @param own_mutex true if caller owns trx_sys_t::mutex */
    void view_close(ReadView*& view, bool own_mutex); //关闭一个read view
    /** Release a view that is inactive but not closed. Caller must own the trx_
```



```

sys_t::mutex.
    @param view      View to release */
    void view_release(ReadView*& view); //是否一个read view

    /** Clones the oldest view and stores it in view. No need to call view_close().
    The caller owns the view that is passed in. It will also move the closed
    views from the m_views list to the m_free list.
    This function is called by Purge to create it view.
    @param view      Preallocated view, owned by the caller */
    void clone_oldest_view(ReadView* view);

    /** @return the number of active views */
    uint size() const;

    /** @return true if the view is active and valid */
    static bool is_view_active(ReadView* view)
    {
        ut_a(view != reinterpret_cast<ReadView*>(0x1));
        return(view != NULL && !(intptr_t(view) & 0x1));
    }

    /** Set the view creator transaction id. Note: This should be set only
    for views created by RW transactions. */
    static void set_view_creator_trx_id(ReadView* view, trx_id_t id);
...
private:
    typedef UT_LIST_BASE_NODE_T(ReadView) view_list_t;

    /** Free views ready for reuse. */
    view_list_t      m_free; //被释放了的read view列表, 准备重用, 避免因创建而浪费时间

    /** Active and closed views, the closed views will have the creator trx id
    set to TRX_ID_MAX */
    view_list_t      m_views;
};

```

### 12.1.2 Read View 快照

读视图本质上就是 MVCC 技术中的一个快照, 所以从 ReadView 类中可以看出几个重点, 一是快照的左右边界, 二是如何判断元组的可见性。

```

/** Read view lists the trx ids of those transactions for which a consistent
    read should not see the modifications to the database. */
class ReadView { //快照, 即活动事务的事务id的范围, 是一个区间。可以看作事务生命长河中的一
    段, 不同的快照就是不同的一段
    /** This is similar to a std::vector but it is not a drop in replacement. It

```

```

is specific to ReadView. */
class ids_t {...};
public:
...
    /** Check whether the changes by id are visible.
    @param[in] id transaction id to check against the view
    @param[in] name table name
    @return whether the view sees the modifications of id. */
    bool changes_visible( //元组的可见性判断。重要函数，详细内容参见12.2节
        trx_id_t id, const table_name_t& name) const MY_ATTRIBUTE((warn_unused_
            result)) {...}

...
private:
...
    /** The read should not see any transaction with trx id >= this
        value. In other words, this is the "high water mark". */
    trx_id_t m_low_limit_id;
    //一个快照，有左右边界，左边界是最小值，右边界是最大值。此变量是右边界

    /** The read should see all trx ids which are strictly
        smaller (<) than this value. In other words, this is the low water mark". */
    trx_id_t m_up_limit_id; //此变量是左边界，小于此值，表示事务发生的更早

    /** trx id of creating transaction, set to TRX_ID_MAX for free views. */
    trx_id_t m_creator_trx_id; //正在创建事务的事务id

    /** Set of RW transactions that was active when this snapshot was taken */
    ids_t m_ids; //快照创建时，处于活动即尚未完成的读写事务的集合

    /** The view does not need to see the undo logs for transactions whose
        transaction number is strictly
        smaller (<) than this value: they can be removed in purge if not needed
        by other views */
    trx_id_t m_low_limit_no;

...
};

```

### 12.1.3 事务与快照

在事务的结构体上，定义有快照这样的成员。这说明快照和事务紧密相关，用以表明此事务与其他事务的生命重叠范围。

```

struct trx_t {...
    ReadView* read_view; /* consistent read view used in the transaction, or
        NULL if not yet set */
...}

```



对于不同的隔离级别，

- ❑ 隔离级别大于等于可重复读：事务块内的所有的 SELECT 操作都要使用同一个快照，此快照是在第一个 SELECT 操作时建立的。
- ❑ 隔离级别小于等于已提交读：事务块内的所有的 SELECT 操作分别创建自己的快照，因此每次读都不同，后面 SELECT 操作的读就可以读到本次读之前已经提交的数据。

## 12.2 可见性判断

### 12.2.1 可见性原则

在一条记录中，记录的一部分结构如表 12-1 所示，其中包含着几个重要的系统字段。

表 12-1 记录的组成表，部分系统列

名称	度 /bytes	意义
DATA_ROW_ID	6	
DATA_TRX_ID	6	本行数据的版本，表明本行记录是被那个事务操作生成的
DATA_ROLL_PTR	7	前一个版本在 UNDO Log 中的位置

DATA\_TRX\_ID 的值，是通过 `trx_write_trx_id()` 函数记载下来的，如图 12-1 表明了此函数被调用的场景，主要是插入、更新操作会把事务号记载到一个记录上的 DATA\_TRX\_ID 表示的位置处，用以表明本条记录是哪个事务操作时生成的，这样就便于与快照中的事务左右边界进行比较（通过调用 `trx_read_trx_id()` 读出记录上的事务 ID，调用 `changes_visible()` 函数比较事务 ID 的值是否在 `[m_up_limit_id, m_low_limit_id]` 区间的左侧、中间、还是右侧，用以决定所读到的记录是否可见）。

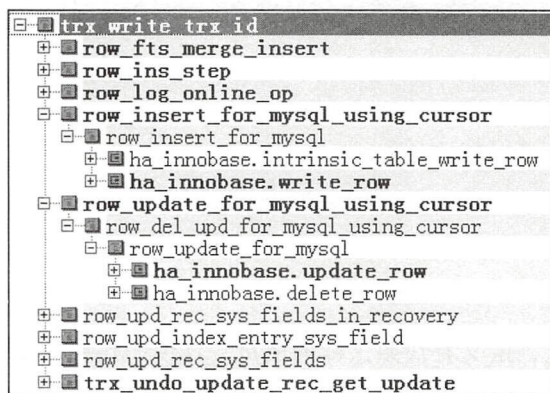


图 12-1 事务操作记录事务 ID 到记录

当一个记录被读取，需要通过可见性原则进行判断，以确认记录是否可以被上层的操作看到。如果不应该被看到，则不可见。

下面先讨论可见性原则。这个原则在快照的数据结构 ReadView 中的 changes\_visible() 方法定义，具体如下：

```
class ReadView { //快照，即活动事务的事务id的范围，是一个区间。可以看作事务生命长河中的一
    段，不同的快照就是不同的一段
    /** This is similar to a std::vector but it is not a drop in replacement.
        It is specific to ReadView. */
    class ids_t {...};
public:
    ...
    /** Check whether the changes by id are visible.
        @param[in] id transaction id to check against the view
        @param[in] name table name
        @return whether the view sees the modifications of id. */
    bool changes_visible( //元组的可见性判断。重要方法。
        trx_id_t id, const table_name_t& name) const MY_ATTRIBUTE((warn_unused_result))
    {
        ut_ad(id > 0);
        if (id < m_up_limit_id || id == m_creator_trx_id) {
            //小于快照的最小边界，则可见。意味着是快照之前发生的事务
            return(true);
        }

        check_trx_id_sanity(id, name);
        //检查事务id的合法性，放到此处判断，暗含着一个意思是：m_up_limit_id
        < trx_sys->max_trx_id

        if (id >= m_low_limit_id) {
            //大于快照的最大边界，一定不可见。意味着是快照之后发生的事务
            return(false);
        } else if (m_ids.empty()) {
            //生成快照时的正发生读写操作的事务集合，如为空，表示不存在读写事务，所以可见
            return(true);
        }

        const ids_t::value_type* p = m_ids.data();
        //正发生读写操作的事务集合，每个快照都有一个其自己对应的这样一个状态集
        return(!std::binary_search(p, p + m_ids.size(), id)); //生成快照时的正发生读
        写操作的事务集合中不包括id指定的事务，则可见；包括则不可见。这时因为在创建快照的时候，
        读写集合中包括某个事务ID表明这个ID对应的事务正在运行中，没有提交，所以不可见。这一点，
        实现的正是读已提交的功能
    }
    ...
}
```

举一个例子：新建一个快照，假设当前事务的事务 ID 为 6，这时，读写事务链表上活跃的读写事务 ID 为 {3,5,6,10}，不包括只读事务。那么调用 view\_open() 创建快照时就会



把 {3,5,10} 存储到当前的视图中的 m\_ids (6 是当前事务的 ID, 不记录到视图中), m\_up\_limit\_id 的值是 3, m\_low\_limit\_id 的值是 10。那么:

- ❑ {3,5,10} 对应的事务所做的修改还处于活动状态即没有提交对当前事务而言都不可见。
- ❑ 小于 3 的事务 ID 如 2 因已经提交 (事务 ID 递增且 2 不处于活动状态表明其已经提交) 对当前事务都是可见的。
- ❑ 大于 10 的事务 ID 是在本事务创建快照之后发生的, 晚于本事务所以对当前事务也是不可见的。

changes\_visible() 方法调用的上下文, 参见图 12-2, 建议读者根据此图自行阅读相关函数, 明确掌握什么时候才进行记录可见性判断。

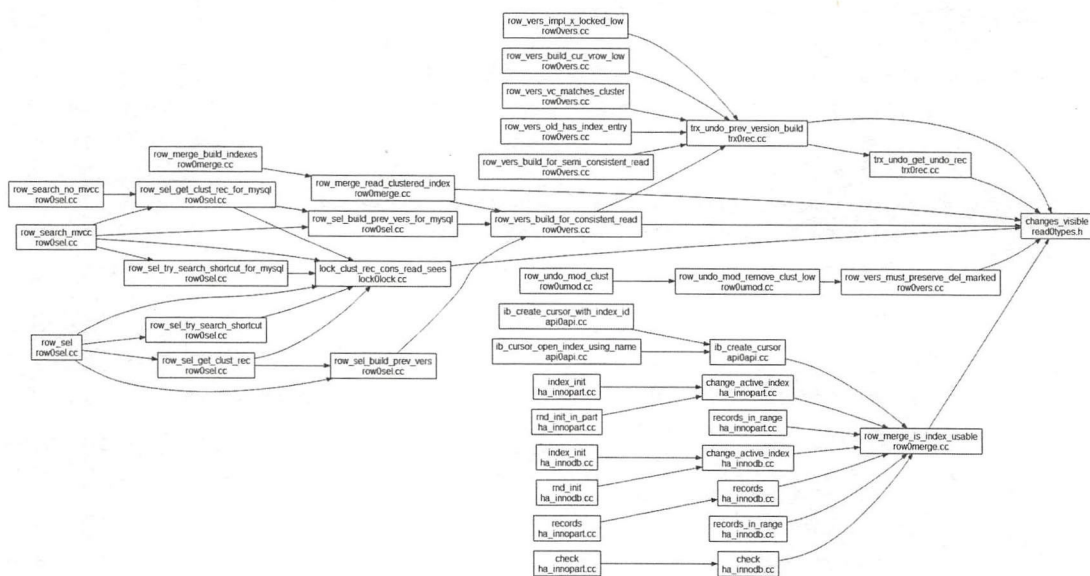


图 12-2 元组可见性判断上下文

下面用一个示例, 来演示可见性判断的上下文调用。

```

/*****
Checks that a record is seen in a consistent read. //在一个一致性读时, 检查记录是否可见
@return true if sees, or false if an earlier version of the record should be retrieved */
bool
lock_clust_rec_cons_read_sees(
    const rec_t*      rec,          /*!< in: user record which should be read or passed
over by a read cursor */
    dict_index_t*     index,        /*!< in: clustered index */
    const ulint*      offsets,      /*!< in: rec_get_offsets(rec, index) */
    ReadView*         view)         /*!< in: consistent read view */
//在指定的快照下, 查看索引index上的rec记录是否可见

```

```

{...
    /* Temp-tables are not shared across connections and multiple transactions
    from different connections cannot
        simultaneously operate on same temp-table and so read of temp-table is
        always consistent read. */
    if (srv_read_only_mode || dict_table_is_temporary(index->table)) {
        //临时表属于某个会话，不能被其他会话操作，因而不存在不一致的问题
        ut_ad(view == 0 || dict_table_is_temporary(index->table));
        return(true);
    }

    /* NOTE that we call this function while holding the search system latch. */
    trx_id_t    trx_id = row_get_rec_trx_id(rec, index, offsets);
    //获取索引上的记录上的事务ID，把此事务ID放到view快照内检查是否可见
    return(view->changes_visible(trx_id, index->table->name));
    //把此事务ID放到view快照内检查是否可见
}

```

### 12.2.2 二级索引的可见性

二级索引上的记录，其上不存在事务 ID，每次更新记录时，将同时更新记录所在的页面 (PAGE) 上的事务 ID (对应 PAGE\_MAX\_TRX\_ID，非记录上的事务 ID)，如果该页面上的事务 ID 对当前事务是可见的，说明此页面上的所有记录都是在快照之前已经完成的事务更新或插入的，所以完全可见，不用找旧版本。否则，表明页面上存在不可见的记录，此时就需要通过 lock\_sec\_rec\_cons\_read\_sees() 函数到对应的聚集索引上查相应的记录，然后利用聚集索引判断可见性 (参见 12.3.3 节)，相关上下文，可以参见 row\_search\_mvcc() 和 row\_search\_no\_mvcc() 函数。

## 12.3 多版本的实现

本节以多版本的实现技术为主要内容，涉及部分 UNDO 日志、PURGE 操作等相关内容，浅尝即止，更多细节，请参阅代码或相关书籍。

### 12.3.1 多版本结构

InnoDB 的 MVCC 技术中的多版本，是根据 UNDO 日志来实现的 (这么说的含义是，InnoDB 实现 MVCC 不仅依据 UNDO，还有如上节讲述的快照和可见性判断原则)。

如表 12-1，在 InnoDB 的记录格式中有两个系统隐含字段，DATA\_TRX\_ID 用来记载一个记录的事务属主，这在 12.2 节讨论过。DATA\_ROLL\_PTR 用来记载本记录的前一个版本在 UNDO 日志中的位置，使用 row\_get\_rec\_roll\_ptr() 函数获取其值即前一个版本的位置。DATA\_ROLL\_PTR 是一个 7 个字节长度，其结构如表 12-2 所示。



表 12-2 DATA\_ROLL\_PTR 结构表

位置	55	54 ~ 48	47 ~ 16	15~0
长度 bit	1	7	32	16
含义	操作类型	回滚段的 ID	UNDO 日志页号	在 UNDO 日志页面上的偏移
可能的值	1 INSERT			
	0 UPDATE			

### 12.3.2 多版本生成

对于更新和删除操作，调用 `trx_undo_page_report_modify()` 函数，把被修改前的记录或被删除前的记录暂存到回滚段。对于插入操作，调用 `trx_undo_page_report_insert()` 函数把插入到索引上的记录的相关信息暂存到回滚段。这几个函数，被 `trx_undo_report_row_operation()` 调用，完成多版本的生成。

对于一个逻辑上的多版本生成过程，其方式如下：

- ❑ 最老的版本，一定是插入操作暂存到 UNDO 日志的版本（对于聚集索引，不是元组的所有字段被暂存到回滚段，而是主键信息被暂存）。
- ❑ 更新操作，把旧值存入 UNDO 日志。同一个记录反复被更新，则每次更新都存入一次旧值（前像）到 UNDO 日志内，如此就会有多个版本。版本之间，使用 `DATA_ROLL_PTR` 指向更旧的版本。由此所有版本构成一个链表，链头是索引上的记录，链尾是首次插入时生成的 UNDO 信息。但如果执行过 `PURGE` 操作，则链表因被清理过可能链尾不再是首次插入时生成的 UNDO 信息。
- ❑ 删除操作，在 UNDO 日志中保存删除标志（用宏 `TRX_UNDO_DEL_MARK_REC` 表示）等信息。
- ❑ 插入或更新操作，可能的因本地更新的可能（in place），导致 `trx_undo_page_report_modify` 函数被多次调用，即插入操作也可能调用此函数（参考 `row_ins_must_modify_rec()` 函数）。

当逻辑上的版本生成后，会调用 `trx_undo_page_add_undo_rec_log()` 函数把相关信息记录到 UNDO 日志上，这是一个物理版本的生成过程。

### 12.3.3 多版本查找

如表 12-2 所示，对于聚集索引，依据 `DATA_ROLL_PTR` 就可以从回滚段中找出前一个版本的记录，并知道此记录是更新操作还是插入操作生成的（表 12-2 中的第 55bit 的标志位）。如果是插入操作生成（第 55bit 的标志位值为 1），则意味着此版本是最原始的版本，即使不可见也不必再继续回溯查找旧版本了（可参考 `trx_undo_prev_version_build()` 函数的实现，因为已经到链表的尾部了，无需再找）。

但是，查找的过程与隔离级别紧密相关：

- ❑ 如果是未提交读隔离级别：根本不去找旧版本，在索引上读到的记录就被直接使用，详情参见 11.3.3 节下的“5. 未提交读的实现”。
- ❑ 如果不是未提交读隔离级别：则需要调用 `row_vers_build_for_consistent_read()` 等函数进入 UNDO 回滚段中根据 `DATA_ROLL_PTR` 进行查找，边找边根据 `changes_visible()` 函数判断可见性，如果可见，则返回（请注意，表 12-1 表明了 UNDO 信息中记载了 `DATA_TRX_ID`、`DATA_ROLL_PTR`，所以可以用 `DATA_TRX_ID` 作为 `changes_visible()` 函数的参数判断可见性，用 `DATA_ROLL_PTR` 查找每一个历史版本）。如图 12-3 所示，可以追溯多版本查找的调用过程。

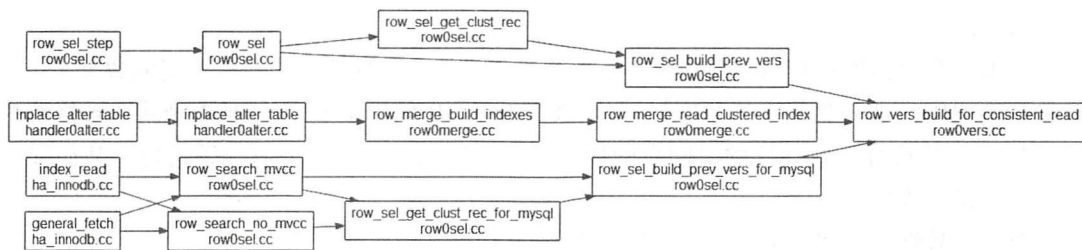


图 12-3 多版本查找的调用过程

### 12.3.4 多版本清理

处于回滚段中的日志，随着事务操作，很多版本陈旧不再需要，就需要进行清理操作。另外，索引页内被打上删除标记的记录也需要物理清理掉。这样的清理工作，由 PURGE 线程负责，通过调用 `trx_purge()` 函数完成。

需要特别注意的，清理的过程也存在一个记录可见性的问题。PURGE 线程会先克隆一个最老的活跃快照（`trx_sys->mvcc->clone_oldest_view()`），所有在此快照创建之前提交的事务所做的数据变更都可以被清理。所以清理过程要用到 12.2 节讲述的可见性原则。

另外，需要注意的一个事情是：长事务（参见 2.5.5 节）影响着清理的起点。所幸只读类型的长事务因 MVCC 技术使得基于封锁的并发控制技术的并发度得到极大提高（参见 2.2.1 节下“5. 锁的并发度问题”），且其不更新数据（如不影响 PURGE 清理），对事务型系统的危害轻微，此类长事务可以认为在 MVCC 技术下被消除了“危害性”。这也说明 MVCC 技术对于事务类型的关系数据库的实现必要性。

至于 PURGE 操作相关的过程逻辑，不再展开描述，请参阅相关代码和书籍。

## 12.4 一致性读和半一致性读

MySQL 提供了一致性读和半一致性读，这些读数据的方式不会对被读的数据进行加锁，有效地提高了并发度。



### 12.4.1 一致性读

MySQL 的官方文档<sup>①</sup>，描述了一致性读如下：

A consistent read means that InnoDB uses multi-versioning to present to a query a snapshot of the database at a point in time. The query sees the changes made by transactions that committed before that point of time, and no changes made by later or uncommitted transactions.

这说明一致性读依赖于 InnoDB 实现的 MVCC 机制。一致性读需要一个时间点，以这个时间点的快照（MySQL 中又称为“read view”，快照是一个活跃的读写事务列表）为读数据的依据，会忽略其他并发事务对于相同数据项的修改，只看到这个快照所限定的范围内的数据。

如果一个数据项被其他并发的事务改变，也不影响一致性读所能获取既定的数据（即一致性读的时间点那一刻确定的数据）。这是因为 InnoDB 使用了 UNDO 日志来获取被修改的数据项之前的符合一致性读的时间点的数据项。这样做得好处是，避免锁的使用从而提高并发事务的并发度。

对于一致性读，实际上包含了三层含义：

- ❑ 在可重复读隔离级别，一致性读是对事务块内的所有查询语句有效。这是事务块的一致性。
- ❑ 在已提交读隔离级别，一致性读是对单个查询语句有效。这是语句级的一致性。
- ❑ 在未提交读隔离级别，读不加锁，读到的是索引上最新的记录。不存在一致性。

这么说的原因可参见 11.3.3 节，在讨论各种隔离级别的实现方式时，我们分析了 InnoDB 是如何在不同隔离级别下使用快照的，实际上说明了不同层级的一致性是如何实现的。

### 12.4.2 半一致性读

MySQL 的官方文档<sup>②</sup>，描述了半一致性读如下：

For UPDATE statements, if a row is already locked, InnoDB performs a “semi-consistent” read, returning the latest committed version to MySQL so that MySQL can determine whether the row matches the WHERE condition of the UPDATE. If the row matches (must be updated), MySQL reads the row again and this time InnoDB either locks it or waits for a lock on it.

这说明半一致性读只适用于更新操作，而且，半一致性读还受限于已提交读隔离级别，代码如下：

```
ha_innbase::try_semi_consistent_read(bool yes) //决定是否可以进行半一致性读
{...
    /* Row read type is set to semi consistent read if this was requested by the
    MySQL and either
```

① <https://dev.mysql.com/doc/refman/5.7/en/innodb-consistent-read.html>

② <https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html>

```
innodb_locks_unsafe_for_binlog option is used or this session is using
READ COMMITTED isolation level. */
if (yes //表扫描或索引扫描
    && (srv_locks_unsafe_for_binlog
        || m_prebuilt->trx->isolation_level <= TRX_ISO_READ_COMMITTED)) {
    //只有隔离级别小于等于已提交读，才可进行半一致性读
    m_prebuilt->row_read_type = ROW_READ_TRY_SEMI_CONSISTENT;
} else { //非表扫描或索引扫描，如全文检索时yes参数值为false
    m_prebuilt->row_read_type = ROW_READ_WITH_LOCKS;
}
}
```

引入半一致性读，在更新操作时，可以减少更新引发的同一行记录的冲突，先允许读到元组，然后交给 MySQL Server 层判断是否满足 WHERE 条件，如果满足再让 InnoDB 去加锁，这样就避免了先加锁后释放的过程，提高了并发度。

## 12.5 本章小结

本章从 MVCC 并发访问控制技术出发，剖析了 InnoDB 对 MVCC 的实现方式，主要讲述了可见性判断原则和多版本的实现内容，从而帮助读者深入理解 InnoDB 使用的 MVCC 并发访问控制的技术。



## 附录

# TDSQL 简介

TDSQL 是腾讯金融云基于 MySQL 的分布式数据库解决方案，旨在解决多副本强一致性、高可用、高性能、分布式、配套设施、安全保障等方面的难题，其在腾讯内部有大量的使用场景，最开始诞生于 2012 年的 TEG 计费平台部，承载了腾讯公司所有数字支付的相关业务，截止目前已承载 200 亿账户数据；2014 年被微众银行（WeBank）选中，作为其核心交易系统的数据库解决方案，以私有云方式交付；2015 年，在腾讯云上正式推出。目前已经为大量金融政企机构提供数据库的公有云及专有云服务，客户覆盖银行、保险、互联网金融、计费、第三方支付、物联网、互联网+、政务等领域。

核心特性：

1) 基于 Raft 的强同步机制：保证强一致性前提下实现高可用，确保数据能实现跨机架、跨 IDC、跨城的数据可靠性，强同步节点故障自动切换，实现数据零丢失。

2) 灵活的全球部署架构：轻松支持异地多活：两 IDC 对等架构；两地三中心架构；两地四中心架构；多地多中心。

3) 数据库内核深度优化：这使得在主备网络延迟 5ms 的情况下，能做到跨 IDC 强同步性能相较于异步同步零损耗；同时 sysbench OLTP TPS 的性能能达到原生 MySQL 版本的 185%。

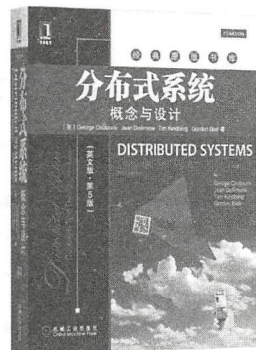
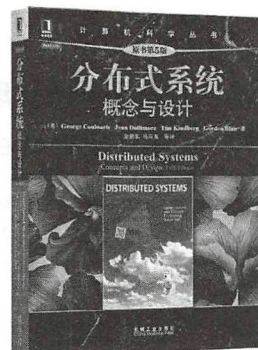
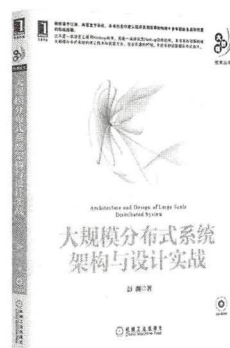
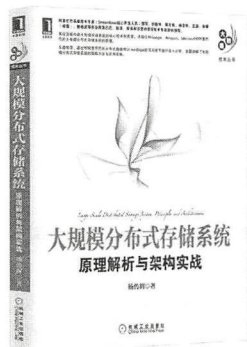
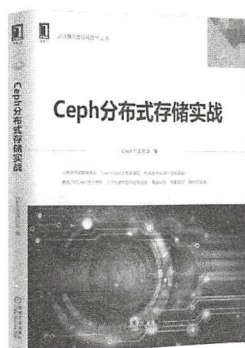
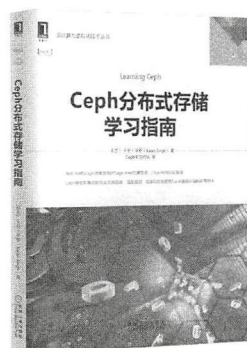
4) 安全增强：在安全方面做了大量优化及增强，包括数据文件加密、SQL 防火墙、SSL 接入、安全审计等。

5) 分布式水平扩展：提供 Auto Sharding 机制，可实现实时在线无缝扩容，确保集群性能和容量呈线性增长；同时提供健壮的 2PC 分布式事务机制，在性能损耗极低的情况下，大大降低了传统业务迁移分布式架构的难度，从此彻底告别数据库中间件。

6) 自动化运营体系：提供完善的运营体系，自动化运营发布平台，机器资源自动管理，智能监控平台及展示，数据库智能诊断等。

7) 完善的配套设施：TDSQL 还提供 Binlog 订阅、冷备系统、多源同步等配套系统，供客户选择。

## 推荐阅读





## 推荐阅读



### 数据库查询优化器的艺术：原理解析与SQL性能优化

作者：李海翔 ISBN：978-7-111-44746-7 定价：89.00元

本书是数据库查询优化领域的里程碑之作，数据库领域泰斗王珊教授亲自作序推荐，PostgreSQL中国社区和中国用户会发起人以及来自Oracle、新浪、网易、华为等企业的数位资深数据库专家联袂推荐。

本书从原理角度深度解读和展示数据库查询优化器的技术细节和全貌；从源码实现角度全方位深入分析MySQL和PostgreSQL两大主流开源数据库查询优化器的实现原理；从工程实践的角度对比了两大数据库的查询优化器的功能异同和实现异同。它是所有数据开发工程师、内核工程师、DBA以及其他数据库相关工作人员值得反复研读的一本书。

### 数据库事务处理的艺术：事务管理与并发控制

作者：李海翔 ISBN：978-7-111-58235-9 定价：99.00元

作者有近20年数据库内核研发经验，曾是Oracle公司MySQL全球开发组核心成员，现在是腾讯的T4级专家。数据库领域的泰斗杜小勇老师亲自为本书作序，数据库学术界的知名学者张孝博士（中国人民大学）、卢卫博士后（中国人民大学）、彭煜玮博士（武汉大学），以及数据库工业界的知名专家盖国强和姜承尧等也给予了极高的评价。

全书共12章，首先介绍数据库事务管理与并发控制的基础理论和工作机制，然后再从工程实践的角度对比和分析了4个主流数据库的事务管理与并发控制的实现原理，最后通过源代码分析了PostgreSQL和MySQL在事务管理与并发控制上的技术架构。

## 作者简介

### 李海翔（网名：那海蓝蓝）

资深数据专家，拥有近20年数据库内核研发经验，曾就职于人大金仓、Oracle公司MySQL全球开发组等，现就职于腾讯TEG计费平台部，T4级专家。中国人民大学工程硕士企业导师。

熟悉PostgreSQL、GreenPlum、MySQL、Informix、CockroachDB等数据库，尤其擅长数据库的查询优化技术、事务处理技术和数据库架构技术。数据库相关工作经历丰富，从事过数据库研发（JDBC驱动、管理工具套机、内核）、数据库测试、技术团队管理、数据库架构设计等多个岗位。

曾获得北京市科学技术进步奖一等奖和腾讯公司级技术突破奖，做过包括863、核高基、工信部、科技部、发改委、北京市科委等多个重大科技项目在内的30多个国家级大型项目。

除本书外，还撰写并出版了本书的姊妹篇《数据库查询优化器的艺术：原理解析与SQL性能优化》，被誉为数据库性能优化领域的经典。



海翔热爱数据库研发，对数据库技术一直抱有一颗坚韧、执着之心，本书是他的经验和思索的体现，值得仔细研读。

——张孝（博士）中国人民大学信息学院副教授

这本书具备较好的深度、广度、新度，这让我十分期待。

——卢卫（博士后）中国人民大学信息学院副教授

本书聚焦于数据库中的事务处理，从原理、主流数据库实现、源码级实现三个角度进行了深度的探讨。尤其是后两个部分的介绍，让本书成为不可多得的有关DBMS事务管理模块内部技术细节的参考资料。

——彭煜玮（博士）武汉大学计算机学院副教授

海翔的著作以独到之角度阐释事务原理与并发控制，以庖丁解牛之刀为广大数据技术从业者剖析出宝贵的关节，实在是让人手不释卷。

——盖国强 云和恩墨创始人/Oracle ACE总监

学习MySQL看姜老师的书，学习优化器和事务处理，就看海翔老师的作品吧。

——姜承尧 腾讯金融支付数据库运营与研发部副总监

作者在数据库内核有多年开发实践与理论研究经验，目前负责腾讯金融分布式数据库TDSQL内核研发工作，旨在解决强一致性、高可用、高性能、分布式、配套设施、安全保障等方面难题，为腾讯内部和外部的海量政企、金融客户的稳定运行保驾护航。



投稿热线: (010) 88379604  
客服热线: (010) 88379426 88361066  
购书热线: (010) 68326294 88379649 68995259

华章网站: [www.hzbook.com](http://www.hzbook.com)  
网上购书: [www.china-pub.com](http://www.china-pub.com)  
数字阅读: [www.hzmedia.com.cn](http://www.hzmedia.com.cn)

上架指导: 计算机/数据库

ISBN 978-7-111-58235-9



9 787111 582359

定价: 99.00元